TURING

黑客与画家

硅谷创业之父Paul Graham文集

Hackers and Painters Big Ideas from the Computer Age

[美] Paul Graham 著

阮一峰 译

O'REILLY®


人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

我们生活在计算机时代，程序员对世界的影响越来越大。那么，他们到底是什么样的人？什么最能触动他们？他们有多大影响力？

不难发现，我们身边的每件东西都逐渐与计算机关联起来了。打字机被计算机取代，手机变成了一台小型的计算机，照相机也是如此。不久以后，电视机和录像机也将会变成计算机网络的一部分。而汽车芯片的运算能力甚至超过了1970年需要一整间屋子才放得下的大型机。信件、百科全书、报纸乃至一些社区商店都正在被互联网取代。下一个改变会是什么？

本书带领我们探究黑客的世界，了解这些人的爱好和动机。Paul Graham妙笔生花，旁征博引历史上的事件，带领读者快速游览了他所谓的“智力西部”，给人以深刻启发。

为什么那些在高中时代默默无闻的孩子，最后却成为了世界上最重要的人物？怎样创业才能成功？理解技术的人与不理解技术的人之间是不是存在一条鸿沟？微软公司会控制互联网吗？怎样才能对付垃圾信息？这些问题涉及教育、技术、管理、道德等，不一而足。

那么，请跟随硅谷创业之父，走进黑客的世界吧！



黑客与画家

硅谷创业之父 Paul Graham 文集

Hackers and Painters: Big Ideas from the Computer Age

[美] Paul Graham 著
阮一峰 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc. 授权人民邮电出版社出版

人民邮电出版社
北 京

图书在版编目（C I P）数据

黑客与画家 : 硅谷创业之父Paul Graham文集 / (美) 格雷厄姆 (Graham,P.) 著 ; 阮一峰译. -- 北京 : 人民邮电出版社, 2011.4（2017.4重印）
书名原文: Hackers and Painters:Big Ideas from the Computer Age
ISBN 978-7-115-24949-4

Ⅰ. ①黑… Ⅱ. ①格… ②阮… Ⅲ. ①计算机网络－安全技术－文集 Ⅳ. ①TP393.08-53

中国版本图书馆CIP数据核字(2011)第033522号

内 容 提 要

本书是硅谷创业之父 Paul Graham 的文集，主要介绍黑客即优秀程序员的爱好和动机，讨论黑客成长、黑客对世界的贡献以及编程语言和黑客工作方法等所有对计算机时代感兴趣的人的一些话题。书中的内容不但有助于了解计算机编程的本质、互联网行业的规则，还会帮助读者了解我们这个时代，迫使读者独立思考。

本书适合所有程序员和互联网创业者，也适合一切对计算机行业感兴趣的读者。

◆ 著　[美] Paul Graham
译　阮一峰
责任编辑　朱　巍
◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子函件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　印刷
◆ 开本：700×1000　1/16
印张：16.5
字数：246千字　2011年4月第1版
印数：75 001－76 500册　2017年4月北京第23次印刷
著作权合同登记号　图字：01-2010-4223号

定价：49.00元
读者服务热线：(010)51095186转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广字第 8052 号

版 权 声 明

O'Reilly Media, Inc.介绍

为了满足读者对网络和软件技术知识的迫切需求，世界著名计算机图书出版机构 O'Reilly Media, Inc.授权人民邮电出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O'Reilly Media, Inc.是世界上在 Unix、X、Internet 和其他开放系统图书领域具有领导地位的出版公司，同时也是联机出版的先锋。

从最畅销的 *The Whole Internet User's Guide & Catalog*（被纽约公共图书馆评为 20 世纪最重要的 50 本书之一）到最早的 Internet 门户和商业网站 GNN，再到第一个桌面 PC 的 Web 服务器软件 WebSite，O'Reilly Media, Inc.一直处于 Internet 发展的最前沿。

许多书店的反馈表明，O'Reilly Media, Inc.是最稳定的计算机图书出版商——每一本书都一版再版。与大多数计算机图书出版商相比，O'Reilly Media, Inc.具有深厚的计算机专业背景，这使得 O'Reilly Media, Inc.形成了一个非常不同于其他出版商的出版方针。O'Reilly Media, Inc.所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O'Reilly Media, Inc.还有许多固定的作者群体——他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O'Reilly Media, Inc.依靠他们及时地推出图书。因为 O'Reilly Media, Inc.紧密地与计算机业界联系着，所以 O'Reilly Media, Inc.知道市场上真正需要什么图书。

保罗·格雷厄姆
其人其事

1964 年，保罗·格雷厄姆（Paul Graham）出生于匹兹堡郊区的一个中产阶级家庭。父亲是设计核反应堆的物理学家，母亲在家照看他和他的妹妹。

青少年时代，格雷厄姆就开始编程。但是，他还喜欢许多与计算机无关的东西，这在编程高手之中是很少见的。中学时，他喜欢写小说；进入康奈尔大学以后，他主修哲学。后来发现哲学很难理解，于是研究生阶段他就去了哈佛大学计算机系，主攻人工智能。

他在这个方向上进展不顺利，因此对学术感到灰心。（但是，作为研究工具的 Lisp 语言，对他日后产生了重大影响。）博士读到一半，他又去哈佛艺术系旁听。拿到博士学位以后，他报名进入罗德岛设计学院暑期班，学习绘画课程，梦想成为画家。

上完暑期班，他去了欧洲，在有 500 年历史的佛罗伦萨美术学院继续学习绘画。第二年，钱花完了，他不得不返回美国，在波士顿的一家创业公司中担任程序员。那时是 1992 年。

此后的两三年，格雷厄姆一直过着一种动荡的生活。他栖身于纽约一间极小的公寓，追求自己的艺术家梦想，但是收入低而且不稳定，日子过得非常窘迫，常常入不敷出，他不得不经常替别人编程，赚取一些生活费。

终于有一天，格雷厄姆觉得不能再这样继续下去了。“我决定不当画家了，首先要彻底解决自己的收入问题。”他后来回忆道。

1995 年的初夏，他找到了读书时认识的朋友罗伯特·莫里斯（Robert Morris），希望合作编写一个软件来赚钱。后者是一个非常聪明的黑客，

曾经在 1988 年编写了历史上第一个蠕虫病毒“莫里斯蠕虫”。

那时正赶上第一家互联网公司网景上市，大量的造势广告在媒体上轮番播出，整个资本市场都为“互联网概念”而疯狂。[①] 格雷厄姆心想，如果网景公司的设想是正确的，未来人们都在互联网上购物，那么必须有人为零售商们开发软件。所以，他决定开发一个搭建网店的软件。

罗伯特·莫里斯此时还在麻省理工学院攻读研究生，只有暑假才有空。于是，格雷厄姆就搬到莫里斯的公寓。两人利用那一段时间写出了产品原型。

一开始，他们的软件完全采用传统模式，即用户首先下载安装，然后在自己的硬盘上做出网店的雏形，最后再上传到服务器。后来，格雷厄姆灵机一动：为什么不让用户通过浏览器直接操作服务器呢！这样就完全省去了安装和上传的步骤。



于是，他们改变方向，决定把软件做成一个互联网程序。这是世界上第一个通过互联网使用的软件。因为这一点，他们就把这个产品起名为 Viaweb。他们放弃开发桌面软件还有另一个原因，那就是两人都没学过如何开发 Windows 程序，并且也不太想学。

他们拿着产品原型找到了格雷厄姆的美术老师的丈夫，他是一位律师，他同意资助 1 万美元。这就是当时的全部资金了。他们用这笔钱买了一台服务器，然后着手将原型程序完善为可以演示的 Demo 版。事情越来越多，两个人忙不过来了，格雷厄姆就问莫里斯：“你的同学之中谁编程最厉害？”回答是特雷弗·布莱克韦尔（Trevor Blackwell）。于是，Viaweb 有了第三个创始人。

八月初，他们做出了可以上线的 Demo 版。十月份，他们来到纽约，向两个天使投资人展示，希望能够筹集到 5 万美元。结果，两个投资人都表示愿意投资，于是他们就拿到了 10 万美元。十二月，Viaweb 正式开始发展客户。

---

① 网景于1995年8月9日在纽约上市，当时它成立还不到16个月，从未赢利过。投资银行事先估计每股仅能卖14美元左右，然而开盘价就是28美元，随后一路攀升，盘中最高价71美元，收盘价为58.25美元。这家原始资本只有400万美元的小公司，只用了一天时间，就成为市值20亿美元的巨人。

1996 年夏天，Viaweb 得到了第二轮天使投资 80 万美元。他们用这笔钱雇了 CEO 和新程序员，还租了新办公室。此前，他们一直在莫里斯公寓的楼上办公，环境十分简陋，总共只有一台电脑，如果有参观者，还必须借一些电脑来“装门面”。现在，Viaweb 终于看上去像一家公司了。也正是从这时开始，有媒体对 Viaweb 进行报道了，格雷厄姆松了一口气，他们终于不必再依靠用户的口口相传来发展业务了。

1996 年圣诞节，他们的客户总数达到了 70 家，其中包括著名摇滚杂志《滚石》。一年后，客户总数增长了 5 倍，公司看起来发展得很好。在这个过程中，他们一直在四处寻找收购者，因为创立 Viaweb 的主要目的就是希望被收购，这样才能快速赚到钱。他们前后经历的收购谈判不下 10 次，但是由于种种原因都失败了。

1998 年，收购终于成功。雅虎以 4900 万美元的价格兼并了 Viaweb，将其改名为 Yahoo! Store，这是雅虎最早的收购行动之一。完成收购的那一天，格雷厄姆把莫里斯拉到哈佛广场的美容店里。因为后者曾经在创业初期说过，如果能从 Viaweb 之中赚到 100 万美元，他就愿意打个耳洞。



被收购之后，格雷厄姆就成为了雅虎的员工，继续从事编程。他在那里工作了一年半，总是感觉很不自在，不适应雅虎的企业文化，用他自己的话说，就是不习惯大公司的官僚环境，他说：“运营创业公司，每天都像在战斗；而为大公司工作，就像在窒息中挣扎。”于是，他选择了辞职。

离开了雅虎，他的生活顿时就空闲了。他开始将自己对于技术和创业的观点写成一系列文章，发表在个人网站上。这些文章受到读者的好评和追捧，访问量不断上升。2004 年，最受欢迎的那部分文章由 O’Reilly 出版社结集出版，取名为《黑客与画家》，这两个词正是格雷厄姆前半生的人生写照。他在前言中写道：“我们生活中的一切，都正在成为计算机。所以，如果你想理解我们目前的世界以及它的未来动向，那么多了解一些黑客的想法会对你有帮助。”

2005 年 3 月，哈佛大学的学生团体“计算机协会”邀请格雷厄姆做演讲。他选择的题目是《如何成立创业公司》。“我对他们说，选择

天使投资人的时候，最好选择那些自己有过创业经验的人。”说完这句话，他发现学生都以期待的眼神看着他，他赶紧补充说：“我不是天使投资人。”

这句话显然说早了。演讲结束以后，他与学生一起喝咖啡聊天，他发现其中有些人很有才干和想法。他不由想到，如果没有天使投资人，Viaweb 根本不可能存在，也就不会有现在的自己。于是，他决定为这些学生创造机会，看看他们能做出什么。

2005 年的暑假，他重新找到了罗伯特·莫里斯（他现在是麻省理工学院计算机系的教师），两人一起举办了一个夏令营，旨在帮助那些有创业念头的大学生成立自己的公司。入选者都将得到他们的悉心指导以及 5000 美元资助。

申请表有 227 份，他们从中挑了 8 个项目。等到夏令营结束的时候，有 4 个项目已经做得很像样了。社会化地理服务网站 Loopt.com 后来得到了两家风投公司共 1300 万美元的投资，截至 2010 年底有 400 万用户；用户聚合的新闻网站 Reddit.com 2006 年被大型出版集团 Condé Nast 以 1000 多万美元的价格收购，目前排名于全美一百大网站之列；移动支付服务网站 TextPayMe 2006 年被亚马逊收购；在线日历网站 Kiko 是该领域的先锋，后来由于谷歌推出同类产品而被迫把源码放到 eBay 上拍卖，卖出了 25.8 万美元。



结果令人鼓舞，格雷厄姆觉得可以把这件事情做下去，将扶植创业公司作为一项事业。于是，他和莫里斯再加上特雷弗·布莱克韦尔和杰西卡·立弗斯通[①]，合伙在硅谷成立了 Y Combinator[②]（Y 运算子，简称 YC）。根据格雷厄姆的设想，它既是一个创业公司的孵化器，也是一个教导员，还是一个与投资人联系的中介。

YC 在每年的一月和六月举办两次训练营，每次为期三个月。通常每次大概有 500 个申请者，他们从中挑出 20 个项目[③]。每个项目将得到 1.1 万美元的启动资金，外加每个项目成员 3000 美元的生活津贴，交换条件

① Jessica Livingston，《创业者》（*Founders At Work*）一书的作者，后来嫁给了格雷厄姆。

② Y Combinator是一个编程术语，意思是创造其他函数的函数。

③ 2010年6月，YC的规模扩展到一次训练营招收了43个创业团队。

是 YC 将拿走该项目 5%的股份[①]。如果项目成功，5%的股份将非常值钱。

YC 的合伙人对每个项目都进行个别辅导，不仅提供项目建议，还灌输方法论和价值观。每个星期四下午，创业者来到 YC 的办公室，与格雷厄姆或者其他某个合伙人见面，报告项目的进展，然后一起讨论如何解决一些棘手的难题。

面谈结束以后，就是聚餐时间。所有人一边吃饭，一边聊天。同时，还有特邀嘉宾与创业者见面。嘉宾往往是技术行业的顶尖名人，比如 Facebook 的创始人马克·扎克伯格（Mark Zuckerberg）、Groupon 的创始人安德鲁·马森（Andrew Mason）等。

三个月的训练营接近尾声时，按照计划，创业者应该拿出一个可以运行的成果。YC 到时会举行“展示日”，让风险投资商与创业者直接见面。创业者在台上展示自己的项目，风投在台下进行评估，有意向的话，双方再进行私下接触。



业界对 YC 毕业生的投资热情在“展示日”表现得一览无遗。最初，只有格雷厄姆的朋友和以前的同事参加，后来要求参加的人越来越多，几乎硅谷所有重要的风投公司和天使投资人都会蜂拥而至，以至于 150 人的会场坐不下，“展示日”不得不延长成 3 天，而每个项目只有两分半钟的自我介绍时间。2010 年 8 月，YC 孵化出的新一届 36 家创业公司，有 30 家得到了风险投资，很多都超过了 100 万美元。

到目前为止，从 YC“毕业”的创业公司共有 200 多家，已经失败的公司不到 20%，远低于 90%的业内平均水平。这些 YC 学员成为新一代硅谷创业公司的主流，他们组成了一张不断壮大的关系网，有人把这些迅速崛起的硅谷新成员称为“YC 匪帮”。

YC 已经改写了创业家和硅谷投资者之间的旧秩序，塑造了创建技术公司的新范式。在科技快速发展、资本不断涌入的大背景下，它使得创业公司规模更小、成本更低、行动更快。

从 2005 年至今，格雷厄姆本人面试过的创业者接近 2000 人，他在某种程度上已经是硅谷的中心人物，有着巨大的影响力。他的文章在美

---

① 这是一个平均值，YC获得股份的最低值是1.4%，最高值是10%。

国创业者中广为流传，年轻的技术人员阅读他的书籍，了解他的思想，讨论他的观点。

这种变化在硅谷引起了不同的反应。2010 年 9 月，著名网志 TechCrunch 的主编米歇尔 • 阿灵顿（Michael Arrington）揭露了一场发生在旧金山某酒吧的秘密聚会，一群显赫的天使投资人抱怨 YC 的势力太过显赫，抬高了风投业的整体估值水平。他们密谋如何压制竞价，把创业公司的估值降下来。另一方面，2011 年 1 月，著名天使投资人尤里 • 米尔纳（Yuri Milner）宣布，将向每一个 YC 的创业项目提供 15 万美元资金，没有任何前提条件，唯一的要求就是如果这些项目有下一次融资，这 15 万美元将获得同等待遇，转为股份。从这两件事上，YC 的风头之劲可见一斑。

保罗 • 格雷厄姆有一套完整的创业哲学，他的创业公式是：

（1）搭建原型

（2）上线运营（别管 bug）

（3）收集反馈

（4）调整产品

（5）成长壮大



首先，他鼓励创业公司快速发布产品，因为这样可以尽早知道一个创意是否可行。其次，他认为一定要特别关注用户需要什么，这样才有办法将一个坏项目转变成好项目。他说："许多伟大的公司，一开始的时候做的都是与后来业务完全不同的事情。乔布斯创建苹果公司后的第一个计划是出售计算机零件，然后让用户自己组装，后来才变成开发苹果电脑。你需要倾听用户的声音，琢磨他们需要什么，然后就去做。"所有学员刚刚来到YC的时候，每人都会拿到一件白色T恤衫，上面写着"Make something people want"（制造用户需要的东西），等到他们的项目得到风险投资以后，又会收到一件黑色T恤衫，上面写着"I made something people want"（我制造了用户需要的东西）。

比起那些令人叫好的创意，格雷厄姆更看重创始人的素质。他说："我们从一开始就认识到，创始人本身比他的创意更重要。"他还认为，小团队更容易成功，创始成员总数最好不要超过三个人。其中一个原因

是，创始人越多，股权越不容易平等分配，容易造成内耗。

格雷厄姆认为，我们正在进入一个创业时代。未来的社会，创业可能成为一种常态，而替别人打工反而成了少见的事情。一方面，创业是最有效的创造财富的方法，对创始人、对投资者、对社会都是如此。“如果拉里·佩奇（Larry Page）和谢尔盖·布林（Sergey Brin）没有创立谷歌，那么他们可能还在某个研究部门工作，写一些不会有多少人使用的代码。但是，他们选择了创业，想一想这样做为全世界增加了多少价值？”另一方面，创业越来越简单了，成本也越来越低。“以前创业很昂贵，你不得不找到投资人才能创业。而现在，唯一的门槛就是勇气。”

格雷厄姆认为，对于科技公司来说，未来充满了机会，前景一片光明。“所有东西都在变成软件。印刷机诞生后，人类写过多少个字，未来就有多少家软件公司。”

作者：阮一峰



译　者　序

你现在拿在手里的是一本非常重要、也非常独特的书。

它的作者是美国互联网界举足轻重、有“创业教父”之称的哈佛大学计算机博士保罗·格雷厄姆（Paul Graham）。本书是他的文集。

书中的内容并不深奥，不仅仅是写给程序员和创业者的，更是写给普通读者的。作者最大的目的就是，通过这本书让普通读者理解我们所处的这个计算机时代。

1968 年至 1972 年期间，美国出版过一本叫做《地球商品目录》（*Whole Earth Catalog*）的杂志，内容从植物种子到电子仪器，无所不包，出版目的据说是要帮助读者“理解整个系统”。多年后，苹果公司的总裁乔布斯盛赞它“有点像印刷版的谷歌”。从某种意义上说，本书也是如此，作者试图从许许多多不同的方面解释这个时代的内在脉络，揭示它的发展轨迹，帮助你看清我们现在的位置和将来的方向。

电子技术的发展，使得计算机日益成为人类社会必不可少的一部分。

每个人日常生活的很大一部分都花在与计算机打交道上面。家用电表是智能的，通信网络是程控的，信用卡是联网的，就连点菜都会用到电子菜单。越来越多的迹象表明，未来的人类生活不仅是人与人的互动，而且更多的将是人与计算机的互动。

想要把握这个时代，就必须理解计算机。理解计算机的关键，则是要理解计算机背后的人。表面上这是一个机器的时代，但是实际上机器的设计者决定了我们的时代。程序员的审美决定了你看到的软件界面，程序员的爱好决定了你有什么样的软件可以使用。

我们的时代是程序员主导的时代，而伟大的程序员就是黑客。

本书就是帮助你了解黑客、从而理解这个时代的一把钥匙。

在媒体和普通人的眼里，“黑客”（hacker）就是入侵计算机的人，就是“计算机犯罪”的同义词。但是，这并不是它的真正含义（至少不是原意），更不是本书所使用的含义。

要想读懂这本书，首先就必须正确理解什么是“黑客”。

为了把这个问题说清楚，有必要从源头上讲起。1946 年，第一台电子计算机 ENIAC 在美国诞生，从此世界上一些最聪明、最有创造力的人开始进入这个行业，在他们身上逐渐地形成了一种独特的技术文化。在这种文化的发展过程中，涌现了很多“行话”（jargon）。20 世纪 60 年代初，麻省理工学院有一个学生团体叫做“铁路模型技术俱乐部”（Tech Model Railroad Club，简称 TMRC），他们把难题的解决方法称为 hack。

在这里，hack 作为名词有两个意思，既可以指很巧妙或很便捷的解决方法，也可以指比较笨拙、不那么优雅的解决方法。两者都能称为 hack，不同的是，前者是漂亮的解决方法（cool hack 或 neat hack），后者是丑陋的解决方法（ugly hack 或 quick hack）。hack 的字典解释是砍（木头），在这些学生看来，解决一个计算机难题就好像砍倒一棵大树。那么相应地，完成这种 hack 的过程就被称为 hacking，而从事 hacking 的人就是 hacker，也就是黑客。



从这个意思出发，hack 还有一个引申义，指对某个程序或设备进行修改，使其完成原来不可用的功能（或者禁止外部使用者接触到的功能）。在这种意义上，hacking 可以与盗窃信息、信用卡欺诈或其他计算机犯罪联系在一起，这也是后来“黑客”被当作计算机入侵者的称呼的原因。

但是，在 20 世纪 60 年代这个词被发明的时候，“黑客”完全是正面意义上的称呼。TMRC 使用这个词是带有敬意的，因为在他们看来，如果要完成一个 hack，就必然包含着高度的革新、独树一帜的风格、精湛的技艺。最能干的人会自豪地称自己为黑客。

这时，“黑客”这个词不仅是第一流能力的象征，还包含着求解问题过程中产生的精神愉悦或享受。也就是说，从一开始，黑客就是有精神追求的。自由软件基金会创始人理查德·斯托尔曼说：“出于兴趣而解

决某个难题，不管它有没有用，这就是黑客。”[①]

根据理查德·斯托尔曼的说法，黑客行为必须包含三个特点：好玩、高智商、探索精神。只有其行为同时满足这三个标准，才能被称为“黑客”。另一方面，它们也构成了黑客的价值观，黑客追求的就是这三种价值，而不是实用性或金钱。

1984 年，《新闻周刊》的记者史蒂文·利维出版了历史上第一本介绍黑客的著作——《黑客：计算机革命的英雄》（*Hackers: Heroes of the Computer Revolution*）。在该书中，他进一步将黑客的价值观总结为六条“黑客伦理”（hacker ethic），直到今天这几条伦理都被视为这方面的最佳论述。

(1) 使用计算机以及所有有助于了解这个世界本质的事物都不应受到任何限制。任何事情都应该亲手尝试。

（Access to computers—and anything that might teach you something about the way the world works—should be unlimited and total. Always yield to the Hands-On Imperative!）

(2) 所有信息应该都是自由的。

（All information should be free.）

(3) 不信任权威，提倡去中心化。

（Mistrust Authority—Promote Decentralization.）

(4) 判断一名黑客的水平应该看他的技术能力，而不是看他的学历、年龄或地位等其他标准。

（Hackers should be judged by their hacking, not bogus criteria such as degrees, age, race, or position.）

(5) 你可以用计算机创造美和艺术。

（You can create art and beauty on a computer.）

(6) 计算机使生活更美好。



① 见理查德·斯托尔曼所著的On Hacking一文，收录于*Free Software, Free Society: Selected Essays of Richard M. Stallman*一书中（CreateSpace，2009），http://stallman.org/articles/on-hacking.html。——译者注

（Computers can change your life for the better.）

根据这六条“黑客伦理”，黑客价值观的核心原则可以概括成这样几点：分享、开放、民主、计算机的自由使用、进步。

所以，“黑客”这个词的原始含义就是指那些信奉“黑客伦理”而且能力高超的程序员。历史上一些最优秀的程序员都是“黑客”。除了上文提到的理查德·斯托尔曼，还包括 Unix 操作系统创始人丹尼斯·里奇和肯·汤普森，经典巨著《计算机程序设计艺术》的作者、斯坦福大学计算机教授高德纳，Linux 操作系统创始人莱纳斯·托沃兹，“开源运动”创始人埃里克·雷蒙德，微软公司创始人比尔·盖茨等。正是黑客把计算机工业推向了更高的高度。

“黑客伦理”的一个必然推论就是，黑客不服从管教，具有叛逆精神。



黑客通常对管理者强加的、限制他们行为的愚蠢规定不屑一顾，会找出规避的方法。一部分原因是为了自由使用计算机，另一部分原因是为了展现自己的聪明。比如，计算机设备的各种安全措施就是最常被黑客破解的东西。史蒂文·利维对这一点有过一段生动的描述：

> “对于黑客来说，关着的门就是一种挑衅，而锁着的门则是一种侮辱。……黑客相信，只要有助于改进现状、探索未知，人们就应该被允许自由地使用各种工具和信息。当一个黑客需要一样东西来帮助自己进行创造、进行探索或者进行修修补补时，他不会自找麻烦，不会接受那些财产专有权的荒谬概念。”

这就是黑客有时会入侵计算机系统的原因，他们的主要目的并不是侵犯别人的利益，这与那些计算机罪犯是不同的。

但是，20 世纪 80 年代初，事情发生了变化。

1983 年，一帮密尔沃基市的青少年黑客入侵了美国和加拿大的一些计算机系统，这件事被广泛报道，同年 9 月 5 日的《新闻周刊》封面报道的标题就是“小心：黑客在行动”，这是历史上主流媒体第一次使用“黑客”这个词。在报道的时候，媒体只注意和强调黑客行为一个很窄的方面：入侵系统。（可能因为这种行为容易引起公众的注意，提升报道的关

注度。）他们把黑客简单定义为入侵系统、破坏安全设施的人。从此，大多数人对于黑客有了错误的看法。同时，那些入侵计算机的程序员也自称“黑客”，使得这个问题进一步复杂化。

杂志、电视剧、电影、小说都对黑客的这种形象大肆渲染。黑客成了反社会的技术高手的代名词，仿佛只要他坐在键盘前，就有一种从事犯罪活动的魔力，可以操纵任何与网络相连的机器，从核弹到车库大门，都在黑客敲打键盘的操作之中被控制。根据这种观点，黑客在最好的情况下是一个没有认识到自己能力的清白的人，在最坏的情况下则是一个恐怖分子。在过去几年中，随着计算机病毒的泛滥，黑客在大众心目中已经成了一个有害的人群。

那些传统意义上的黑客不认同这样使用“黑客”这个词。他们认为，历史上确实有一些正直的黑客，为了亲自了解系统，做过违反法规的入侵举动。但是，那些人并没有恶意，而且从一开始恶作剧就是黑客文化的一部分，仅仅由此推断入侵和破坏系统就是黑客文化的实质完全是错误的。真正的黑客致力于改变世界，让世界运转得更好。媒体对黑客的定义未免过于片面。



为了澄清“黑客”这个概念，他们提出只有传统意义上的黑客才能被称为 hacker，而那些恶意入侵计算机系统的人应该被称为 cracker（入侵者）。这个观点已经在程序员社区中得到普遍认同。

本书正是在这个意义上使用“黑客”这个词。在本书中，“黑客”就是指最优秀的程序员，而不是入侵计算机系统的人。

为了帮助读者理解黑客，全书 15 章可以大致分成三个部分。

- 第一部分从第 1 章到第 4 章，解释了黑客是如何成长的以及他们看待世界的一些观点。
- 第二部分从第 5 章到第 9 章，解释了黑客怎样做出自己的成果，这些成果又是怎样对全世界产生了影响。
- 第三部分从第 10 章到第 15 章，解释了黑客的工具（编程语言）和工作方法，这是黑客文化的基础和核心。

作者想让公众了解，黑客并不神秘，更不是技术怪人。《黑客与画家》

这个书名就是在提示应该把黑客与画家当作同一种人看待。和画家一样，黑客只是怀有一门特殊手艺、有创造天赋的普通人。这个书名还有另一层含义，即编程是一种艺术创作，黑客就是艺术家，开发软件与画家作画、雕塑家雕刻、建筑师设计房屋并没有本质不同。

总之，这是一本帮助你理解这个时代的书。作者想教给你的其实是新思想。读完以后，你看待世界的眼光很可能会完全不同了。如果你想在 21 世纪立足，理解这一次新的技术革命，做一个掌握自己命运的成功者，我建议你读这本书。

翻译过程中，我已经尽了最大努力，力求把原文忠实、清晰地译成中文。本书的一些章节中作者谈论的都是计算机行业的专业问题，但是他又希望让普通读者看懂，试图用口语化、生活化的语言解释专业概念，我个人感觉效果不太理想，反而使得行文稍显冗余和模糊，这一点提醒读者注意。总的来说，这是一本计算机行业的经典著作，2004 年在美国出版后一直畅销不衰，深受好评，很多读者都被作者的远见卓识所折服。



最后，本书第 13 章涉及 Lisp 语言的部分曾请田春帮助校译，在此表示衷心感谢。

阮一峰

2010年12月25日，写于上海

献给我的母亲

致 读 者

本书各章的内容相互独立，不必按顺序阅读，你完全可以跳过不感兴趣的章节。如果遇到不理解的技术术语，请参考书后的术语表，或者在第 10 章中寻找答案，那一章解释了与软件有关的大量概念。

我们还要遗憾地告诉读者，微软公司的公关顾问在读完第 5 章后，不同意授权我们使用他们手中的比尔·盖茨照片。我们因此要感谢新墨西哥州阿尔伯克基（Albuquerque）市警察局提供第 89 页上的替代照片[①]。

若想了解作者的更多见解，请登录 www.paulgraham.com。

---

① 1977年，22岁的比尔·盖茨因为无证或超速驾驶，在阿尔伯克基市被警察逮捕，因此在警察局留下了档案照片。——译者注

前　　言

本书尝试解释计算机世界里发生了什么事，所以，它不仅仅是写给程序员看的，也适合所有人。比如，第 6 章讲如何致富，我相信这是所有读者普遍感兴趣的内容。

你可能注意到了，过去三十年中，很多赚到大钱的人都是程序员，比如比尔·盖茨、史蒂夫·乔布斯、拉里·埃里森。为什么？为什么是程序员，而不是土木工程师，或者摄影师，或者精算师？第 6 章将告诉你答案。

软件带来财富，仅仅代表了大趋势的一面而已。这种大趋势就是本书的主题。我们的时代是计算机时代。以前，人们曾经认定这个时代应该是太空时代或者原子时代。但是事实证明，它们只是公关公司发明的概念。计算机对人类生活的影响远远超过了太空航行或者原子技术的影响。

我们生活中的一切，都正在成为计算机。打字机被计算机取代了，电话也变成了计算机，照相机亦是如此。很快，电视机也将变成计算机。当今小轿车所具备的计算能力比1970年占满一间屋子的大型计算机还要强。信件、百科全书、报纸，甚至本地的小店，都正在被互联网取代。所以，如果你想理解我们目前的世界以及它的未来动向，那么多了解一些黑客的想法会对你有帮助。

黑客？那不是侵入他人计算机的人吗？在外行人看来，这个词的意思就是这样。但是在计算机世界中，黑客指的是专家级程序员。因为本书的目的是解释真实的计算机世界是怎么一回事，所以我决定冒着被误解的风险，按照行业内的定义使用这个词。

本书的前几章回答了一些大家可能都想过的问题。怎样创业才会成功？技术是否造成了技术人员与普通人之间的隔阂？程序员到底在做些

什么？为什么那些读高中时普普通通的学生，最终却摇身一变成为世界上最有影响力的人士？微软公司会控制互联网吗？怎样才能对付垃圾邮件？

本书后面几章谈的是大多数非计算机行业的人士没有想过的问题——编程语言。为什么普通人要去关心编程语言？因为如果你想了解黑客，就必须懂一点编程语言。这就好比回到 1880 年，如果你想理解技术发展，就必须懂一点蒸汽机。

计算机程序只是文本而已。你选择什么语言，决定了你能说什么话。编程语言就是程序员的思维方式。

因此很自然，编程语言对程序员的思想有巨大的影响。你从他们写的软件中就可以看出来。旅游网站 Orbitz 成功打入了竞争激烈的网络订房订票市场。该市场原先被两大巨头主宰，一个是微软公司，另一个是拥有数十年电子预定服务经验的 Sabre。Orbitz 是怎么从它们手中抢到市场的？最主要的原因就是它使用了一种更好的编程语言。



根据使用的语言不同，程序员往往会被分成不同的派别。人们区分程序员甚至不是看他们写了什么程序，而是看他们使用什么语言。所以，声称一种语言优于另一种语言被认为是不礼貌的行为。但是，没有一个编程语言的设计者会相信“不同的语言各有千秋”这种文绉绉的客套话。我将直言不讳地说出自己对于编程语言的看法，这也许会令很多人不快，但是如果想要理解黑客，我真的觉得没有其他更好的方法。

有些读者可能不理解第 3 章的内容。这一章与计算机有什么关系？事实是黑客很在乎言论自由。Slashdot（它是黑客世界的《纽约时报》）有一个专栏讨论这个问题。我想 Slashdot 的大多数读者都认为重视言论自由是天经地义的事情。不过《飞机与飞行员》（*Plane & Pilot*）杂志就肯定不会有一个这样的栏目。

为什么黑客那么在乎言论自由？我认为，部分原因在于，革新对于软件行业实在是太重要了，而革新和异端实际上是同一件事。优秀的黑客养成了一种质疑一切的习惯。这是肯定的，因为如果你不得不同一台机器打交道，而这台机器全部由文字组成，像机械式手表一样复杂，并且规模大出 1000 倍，那么你也会养成这种习惯的。

此外，我还认为，行为怪异的人和愤世嫉俗的人比普通人更可能成为黑客。计算机世界就像是智力世界的大西部，在那里没有你不敢想的事情，只要你愿意承担冒险后果。

如果我成功实现了自己对这本书的设想，那么它也将是一片智力的大西部。我不希望你带着某种压迫感来阅读此书，心里想着："没办法，这些书呆子看上去正在接管世界。我最好能懂一点他们正在干的事情，这样就不会被他们整出来的下一个东西唬到了。"如果你喜欢思考，阅读此书应该会带给你很多乐趣。虽然黑客从外表看上去一般都是呆呆的，但是他们的大脑内部却是一个有趣得让你吃惊的地方。

写于马萨诸塞州坎布里奇

2004 年 4 月



目　录



1

为什么书呆子不受欢迎

初中时，我和好友里奇画了一张学校食堂的餐桌分布图，每张桌子都标上了人气指数。这件事的难度并不高，因为选择坐在一起吃午饭的人，受瞩目程度往往都很接近。我们把所有桌子从 A 到 E 分成五等。坐在 A 桌的人不是校足球队的成员，就是啦啦队的成员。而 E 桌的人好像都有一点轻微的唐氏综合征[①]，那时我们管这种症状叫“弱智”。



我和里奇在 D 桌。一般来说，只要你不是外貌猥琐，这就是你能分到的最低档次的桌子了。我们把自己的桌子列为 D 桌，倒不是故意谦虚，而是无法自欺欺人。因为学校里每个人的级别都是明摆着的，我们想骗自己也骗不了。

我后来认识很多人，读书的时候都被称为书呆子。从他们身上我发现，“书呆子”与“高智商”有强烈的正相关关系。而这些人在中学里都是不受欢迎的学生，你越喜欢读书，就越不受别人的欢迎，因此“书呆子”和“受欢迎”之间，有一种更强烈的负相关关系。这样看来，“高智商”似乎导致了你不受欢迎。

为什么会这样？要是你眼下还在读中学，你一定会觉得这个问题很蠢。无可争议的事实就是，除了这样以外，似乎很难想象还能有什么别的结果。但是，的确会出现别样的情况。比如，在小学里，聪明的学生就没有受到排挤。再比如，毕业后踏上社会，聪明也不是一件坏事。而且，据我所知，在大多数国家，事情也没有如此严重。只有在典型的美

① 唐氏综合征（Down’s Syndrome）是一种先天性疾病，主要指幼儿的智力和体格发育迟缓，特征为低智商、身材矮小、表情呆滞。1866年，英国医生唐·约翰·朗顿首先发现了这一疾病。——译者注

国中学，做一个聪明的学生才是一件麻烦事，你的日子会很不好过。这究竟为什么？

解开这个谜的关键是把问题换一种提法。为什么聪明的小孩没有让自己变得受欢迎？如果他们真的很聪明，为什么找不到受欢迎的诀窍呢？他们在标准化测试中表现得这么好，为什么就不能在这方面也大获成功呢？

有一种观点认为，其他小孩妒忌聪明学生，所以聪明的学生不可能受到欢迎。我倒希望这种解释是对的。回想起来，要是初中里真的有人妒忌我，那么他们一定费了很大力气才把这种妒忌隐藏得无法发现。而且，在任何情况下，如果聪明真的令他人妒忌，这反而会招来女生。因为女生喜欢被其他男生妒忌的男生。

在我就读过的学校，聪明根本就是无足轻重的一样东西。同学们既不看重它，也不唾弃它。如果别的事情都相同，那么大家还是愿意自己变得聪明一点，因为这总比做个笨人好。但是总的来说，智力在大家心里的分量远远不如相貌、魅力和运动能力的分量重。

所以，如果智力本身与“受欢迎”无关，为什么聪明的小孩一直不受同龄人的欢迎呢？我认为，答案就是他们真的不想让自己受欢迎。

如果当时有人告诉我这个答案，我一定会嘲笑他。在学校里不受欢迎，你的日子就很难过，有人甚至因此自杀。所以，要是你跟我说，是我本人不想受欢迎，那就好比你在说，我在沙漠里快渴死了，却又不想喝水。别搞错了，让自己更受欢迎，这才是我要的。

但是事实上，我并不是那么强烈地渴望这个。我更想追求的是另一件事情——聪明。这不仅仅意味着在学校得到好成绩（虽然某种程度上这也挺重要）。我真正想要的是，能够设计奇妙的火箭、写出漂亮的文章、理解编程原理。一句话，我想要做伟大的事情。

那时，我从没试过将梦想分门别类、一一排序。要是我真做了，就会一眼看出聪明是排在最前面的。如果有人许诺，使我一举成为全校最受瞩目的学生，代价是从此智力平庸（请允许我在这里自命不凡），我是绝不会答应的。



虽然“书呆子”饱尝不受欢迎之苦，但是为了解除痛苦而让他们放弃“聪明”，我想大多数人是不会愿意的。对他们来说，平庸的智力是不可忍受的。不过，要是换了别的孩子，情况就不一样了，大多数人会接受这笔交易。对于很多人来说，这反而是更上一层楼的机会。即使是那些智力排名在前 20%的学生（我在这里假设智力可以测量，那时的人们似乎都相信这一点），谁不愿意用 30 分的成绩换来别人的友爱和钦佩？

我认为，这就是问题的根源。“书呆子”的目标具有两重性。他们毫无疑问想让自己受欢迎，但是他们更愿意让自己聪明。“受欢迎”并不是你在课后时间随便做一做就能实现的，尤其是在美国的中学中，在这里，所有人为了个人魅力都会进行激烈竞争。

文艺复兴时期的代表人物阿尔伯蒂[①]有一句名言：“任何一种艺术，不管是否重要，如果你想要在该领域出类拔萃，就必须全身心投入。”[②]我很想知道，世界上是否还有人比美国的中学生在塑造个人魅力方面更加孜孜不倦、精益求精。相比之下，美国海军的海豹突击队（Navy SEALS）成员和神经外科的住院医生[③]都成了懒汉。他们至少还有假期，有些人甚至还有业余爱好。但是，一个美国的青少年在醒着的每一分钟，都在琢磨怎样才能更受欢迎，一年 365 天，天天如此。



我并不是说这些青少年有意这样做。某些人确实是，小小年纪就成为了权谋家，但是大多数人不是。我在这里真正想要表达的是，青少年每时每刻都想融入群体之中。

举例来说，青少年往往很关注服饰。他们这样做并不是有意让自己赢得大众，他们的目的只是要穿得好看。但是穿给谁看呢？无非就是其他小孩。同伴的意见成为他们判别事物的标准，这不仅体现在衣着上，还体现在他们做的几乎每一件事情上，就连走路的姿势也不例外。所以，

① 阿尔伯蒂（Leon Battista Alberti，1404年2月18日—1472年4月20日）是文艺复兴时期意大利的一位通才，在建筑、文学、艺术、宗教、哲学、语言学、密码学等领域都有重大建树，被视为“文艺复兴人”（Renaissance Man，指多才多艺的人物）的典型代表。
——译者注

② 出自Leon Battista Alberti所著的*The Use and Abuse of Books*（《论书籍的正常使用与滥用》）1999年出版。

③ 这两个职业都以工作时间长、负荷高而著称。——译者注

他们为了把所有事情“做对”，所付出的任何努力，不管是有意还是无意，实际上都等同于努力在使自己变得更受欢迎[①]。

书呆子没有认识到这一点。他们没有认识到“受欢迎”需要付出如此之多的努力。一般来说，对于那些高度困难的领域，只有身处其中的人，才能意识到成功需要不间断（虽然未必是自觉的）付出。举例来说，大多数人似乎认为，绘画能力与生俱来，画家就像高个子一样，是天生的。事实上，大多数“会画”的人，本身就很喜欢画画，将许许多多时间投入其中，这就是为什么他们擅长画画的原因。同样的，受欢迎也不是天生的，而是要你自己做出来的。



图1-1　Gateway高中的象棋俱乐部，1981年。左上角的人就是我



书呆子不受欢迎的真正原因，是他们脑子里想着别的事情。他们的注意力都放在读书或者观察世界上面，而不是放在穿衣打扮、开晚会上面。他们就像头顶一杯水来踢足球，一边踢球，一边拼命保持不让水洒

①“受欢迎”的英语单词是popular，这个词还有另一个意思，“大众化的，多数人的”，比如popular support（民意的支持）。此处使用了双关语，作者既是说青少年的行为目的是为了得到同伴的关注和称赞，也是说青少年这样做是为了与群体保持一致。理解这一点，对理解这篇文章非常重要。后文中有些地方的“受欢迎”，用的也是这个双关的含义，下文不再一一说明。——译者注

出来。其他人都在一门心思玩足球，遇到这样的对手，自然能够毫不费力地击败，并且心里还奇怪，对方怎么如此无能。

就算书呆子心里想着变得与其他小孩一样受欢迎，做起来却是难上加难。因为那些受欢迎的小孩从小就在琢磨如何受欢迎，打心底里追求这个。但是，书呆子从小琢磨的却是如何更聪明，心底里也是这样追求的。这都是受父母的影响，书呆子被教导追求正确答案，而受欢迎的小孩被教导讨人喜欢。

到目前为止，我一直把“聪明的学生”和“书呆子”当作同义词，好像它们完全可以换着用。事实上，只有在我上面谈到的这种环境中才能这样使用。所谓“书呆子”，其实只是指这个人的社交技能不够强。但是，你到底需要多“强”的社交技能，取决于你所处的环境。在普通的美国学校中，成为“强人”的标准高得吓人（或者至少是十分特别），即使你不是很“呆”的人，相比之下，也只能算是呆子了。



仅有很少的聪明小孩，能够分配出足够的心思，去关心如何让自己受欢迎。他们往往碰巧还具有俊俏的外表、运动员的体格，或者受人瞩目的兄弟姐妹。不然的话，你就别无选择，只能成为书呆子了。这就是为什么聪明的小孩在青少年时期，比如 11 岁到 17 岁，有着一生中最糟的人生经历。人生的这个时期比其他任何时期更多地受到你的受欢迎程度的影响。

11 岁以前，小孩的生活由家长主导，其他孩子的影响有限。孩子们不是不关心小学里其他同学的想法，但是后者不具有决定性影响。小学毕业以后，这种情形开始发生变化。

到了 11 岁左右，孩子们逐渐把家庭生活当作寻常事了。他们在同伴中开辟了一个新的世界，并认为那个世界才是重要的，比家里的世界更重要。实际上，如果他们在家里与父母发生冲突，反而能在那个新的世界中挣得面子，而他们也确实更在乎那个世界。

但是，问题在于，孩子们自己创造出来的世界是一个非常原始的世界。如果你听任一群 11 岁的孩子自行其是，最后就会发生小说

《蝇王》[1]中的情景。我同许多美国孩子一样，在学校里就被要求阅读此书。这可能不是巧合，有人可能希望借此向我们指出，我们这些孩子就是“野蛮人”，我们自己创造的世界是一个残酷和愚蠢的世界。对当时的我来说，领会这些意思有点太难了。虽然这本书读起来让人觉得完全可信，但是我一点儿没有读出文字背后的意思。当年，他们还不如直截了当告诉我们，我们这些孩子就是“野蛮人”，我们的世界愚不可及。

要是不受欢迎仅仅意味着不受到关注，书呆子们可能觉得还能忍受。不幸的是，在学校里不受欢迎等同于被歧视和被欺负。

为什么会被歧视和欺负？所有现在还在学校里读书的人可能会又一次觉得，怎么会有人问出这么蠢的问题。怎么可能会有其他结果呢？当然会有其他结果。一般来说，成年人就不会去欺负书呆子。为什么小孩子会这样做呢？



一部分原因是，青少年在心理上还没有摆脱儿童状态，许多人都会残忍地对待他人。他们折磨书呆子的原因就像拔掉一条蜘蛛腿一样，觉得很好玩。在一个人产生良知之前，折磨就是一种娱乐。

孩子们欺负书呆子的另一个原因是为了让自己感到好受一些。当你踩水的时候，你把水踩下去，你的身体就会被托起来。同样，在任何社会等级制度中，那些对自己没自信的人就会通过虐待他们眼中的下等人来突显自己的身份。我已经意识到，正是因为这个原因，在美国社会中底层白人是对待黑人最残酷的群体。

但是我认为，孩子们欺负书呆子的主要原因也与追求“受欢迎”的心理有关。怎样才能让自己更受欢迎？个人魅力只是很小的一方面，你应该更多地考虑如何结盟。秘诀就是不停地设法使自己与其他受欢迎的人变得关系更密切。没有什么比一个共同的敌人更能使得人们团结起

---

① 《蝇王》（*Lord of the Flies*）是英国作家威廉·戈尔丁（William Golding，1911—1994）发表于1954年的小说，讲述了一群6~12岁的男孩被困在一个荒岛上的故事。起初，所有人为了生存，互相帮助；后来逐渐分裂成两派，互相残杀，一派象征着文明与秩序，另一派象征着野蛮和混乱。作者通过这个故事，表达了每个人心中都存在着黑暗的力量。此书出版后引起轰动，成为当年最畅销的书籍，被认为是英国的经典文学作品之一，西方各国学校将其列为学生必读书。1983年，威廉·戈尔丁因为此书获得了诺贝尔文学奖。——译者注

来了。

这就好比一个政客，他想让选民忘记糟糕的国内局势，方法就是为国家找出一个敌人，哪怕敌人并不真的存在，他也可以创造一个出来。一群人在一起，挑出一个书呆子，居高临下地欺负他，就会把彼此联系起来。一起攻击一个外人，所有人因此就都成了自己人。这就是为什么最恶劣的以强凌弱的事件都与团体有关的原因。随便找一个书呆子，他都会告诉你，一群人的虐待比一个人的虐待残酷得多。

如果说其中还有一丝安慰，那就是书呆子不妨记住，这种虐待不是针对个人的。一群孩子成群结伙地欺负你，那并不是因为你做错了什么，而是因为这一伙人需要找一件事情一起干，这就好像一群人成群结伙地去打猎一样。他们实际上并不恨你，他们只是需要一个共同的目标。

因为书呆子是不受欢迎的，处在学校的底层，所以全校学生都把书呆子当作一个可供欺负的安全目标。如果我没记错的话，最受欢迎的孩子并不欺负书呆子，他们不需要靠踩在书呆子身上来垫高自己。大部分的欺负来自处于下一等级的学生，那些焦虑的中间层。



麻烦的是，这样的人数量庞大。受欢迎的学生的分布并不是金字塔形的，而是像一个倒放的梨子，底部逐渐收窄。最不受欢迎的人数相当少。（我相信，在我们画的餐桌分布图中，称得上D桌的，只有我所在的那一桌。）所以，想要欺负书呆子的人比被欺负的书呆子多得多。

与不受欢迎的小孩保持距离，可以为你加分；那么与他们关系密切，就会为你减分。我认识的一位女性说，她在高中时对书呆子有好感，但是害怕被人看到她与书呆子说话，因为其他女孩会因此取笑她。不受欢迎是一种传染病，虽然善良的孩子不会去欺负书呆子，但是为了保护自己，也依然会与书呆子保持距离。

难怪聪明的小孩读中学时往往是不快乐的。他们有其他的兴趣，没有多余的精力用来使自己更受欢迎。你在其他地方有所得，就会在这个地方有所失。不受欢迎使得书呆子成为全校攻击的目标。令人惊奇的是，这种噩梦般的情景并非出自任何有预谋的恶意，而仅仅因为这个特殊的环境。

对我而言，最糟糕的日子是初中。孩子们内部的世界刚刚形成，一切都很严酷，聪明的孩子与普通的孩子，人与人之间的差异慢慢开始显露。几乎每一个和我讨论过的人都同意，人生的最糟糕时期是在 11 岁到 14 岁。

在我读的学校，八年级的时候（也就是我 12~13 岁的那一年）曾经发生过一件引起轰动的事情。有一个老师在等校车的时候，偶然听到一群女生在议论某个书呆子如何被欺负，她深感震惊，第二天就向全班发表了言词恳切的呼吁，请大家不要如此残忍地对待同学。

她的呼吁并没有产生实际效果。那时，最触动我的是，她居然对这件事感到震惊。这是不是意味着她以前对此一无所闻？她觉得这一切是不正常的？

没错，成年人不知道孩子们内部发生的事。认识到这一点很重要。在抽象意义上，成年人知道孩子的行为有时是极端残酷的，这正如我们在抽象意义上知道贫穷国家的人民生活极端艰难。但是，像所有人一样，成年人不喜欢揪住不放这种令人不快的事实。你不去埋头探寻，就不会发现具体的证据，就会永远以为这件事是抽象的。



公立学校的老师很像监狱的狱卒。看管监狱的人主要关心的是犯人都待在自己应该待的位置。然后，让犯人有东西吃，尽可能不要发生斗殴和伤害事件，这就可以了。除此以外，他们一件事也不愿多管，没必要自找麻烦。所以，他们就听任犯人内部形成各种各样的小集团。根据我读到的材料，犯人内部的关系是扭曲、野蛮、无孔不入的。处在这种人际关系的最底层可不是好玩的事。

总体上看，我就读的学校与上面说的监狱差不多。校方最重视的事情，就是让学生待在自己应该待的位置。与此同时，让学生有东西吃，避免公然的暴力行为，接下来才是尝试教给学生一些东西。除此以外，校方并不愿意在学生身上多费心思。就像监狱的狱卒，老师们很大程度上对学生是放任自流的。结果，学生就像犯人一样，发展出了野蛮的内部文化。

可是，为什么离开学校以后，真实的世界却能友好地对待书呆子呢？

答案似乎很简单，因为那是成年人的世界，他们都成熟了，不会把书呆子挑出来欺负。不过，我觉得这不是主要答案。监狱里的成年人不也照样以强凌弱吗？而且很显然，上层社会的阔太太之间也是如此，在曼哈顿的某些地方，女性之间的交往听来就像高中时代的延续，同样充满了各种算计和勾心斗角。

我认为，真实世界的关键并非在于它是由成年人组成的，而在于它的庞大规模使得你做的每件事都能产生真正意义上的效果。学校、监狱、上流社会的女士午餐会，都做不到这一点。这些场合的成员都好像关在封闭的泡沫之中，所作所为只对泡沫内部有影响，对外部没有影响。那么很自然地，这些场合就会产生野蛮的做法。因为它们不具备实际功能，所以也就无所谓采用的形式[①]。

当你所做的事情能产生真实的效果，那就不仅仅是好玩而已了，发现正确的答案就开始变得重要了，这正是书呆子的优势所在。你马上就能联想到比尔·盖茨。他不善于社交是出了名的，但是他发现了正确的答案，至少从收入上看是如此。

真实世界的特点是，它极其庞大。如果总体足够大，即使是人数最少的少数派，只要聚集在一起，也能产生可观的力量。在真实世界中，书呆子在某些地方聚集起来，形成自己的社区，智力因素成为那里最被看重的东西。有时，这种因素甚至会以相反的形式表现出来，特别是在大学的数理学系，书呆子甚至会夸大笨拙，以显示自己的聪明。约翰·纳什[②]非常钦佩诺伯特·维纳[③]，就学维纳的样子，经过走廊的时候都用手扶着墙走路。

我 13 岁的那一年，对世界的全部认识，就是身边看到的一切。我以

① 这一句的原文是They have no function for their form to follow，指的是建筑学和工业设计的Form follows function（功能决定形式）原则，即建筑物或工业产品的外在形式为其功能服务，参考网址http://en.wikipedia.org/wiki/Form_follows_function。——译者注

② 约翰·纳什（John Nash，1928—），美国著名数学家，因其对博弈论的突出贡献而获得1994年的诺贝尔经济学奖。纳什从小性格孤僻，不合群，读大学期间以行为古怪而闻名，30岁时就患上了严重的精神分裂症。电影《美丽心灵》讲述的就是他的故事。——译者注

③ 诺伯特·维纳（Norbert Wiener，1894—1964），美国数学家，在电子工程等方面做出了巨大贡献，是随机过程研究的先驱。——译者注



为，我所经历的种种扭曲的事件就是世界的样子。看上去，这是一个残酷的世界，也是一个乏味的世界，我不太肯定哪一个更糟一些。

因为我在这个世界中过得并不好，我觉得一定是自己什么地方做错了。我没有意识到，作为书呆子，我不适应周围环境，某种程度上正说明我领先了一步。书呆子已经在思考的东西，正是真实世界看重的东西。他们与别人不一样，不把所有时间用来玩一种耗尽全力但又毫无意义的游戏。

我们的感受，有点像一个被重新塞进中学的成年人。他不知道穿什么衣服，听什么音乐，用什么暗语。在别的孩子眼里，他就像一个彻头彻尾的外星人。不过，成年人很清楚不用在乎别人怎么想，我们就没有这种自信了。

许多人似乎认为，聪明的小孩在人生的这个阶段应该与“正常的”小孩待在一起。也许吧。但是，至少在某些情况中，书呆子感到不适应的真正原因是其他人都是疯子。我记得读高中的时候，有一次在体育馆观看校运动队的出征大会，啦啦队把对手的模拟像扔到看台上，观众一哄而起，把它撕成碎片。我感到自己仿佛是一个探险家，正在目睹某种奇特的部落仪式。



如果能回到过去，我会向 13 岁的我提供一些建议，主要告诉他要昂起头看世界。我在那个年纪根本不知道这一点，而我身边的世界又虚假得像奶油夹心蛋糕一样。不仅是学校，整个小镇都很虚假，不像真实的世界。为什么人们要搬到郊区去住？为了养育下一代！难怪郊区生活是如此地乏味和贫瘠。整个镇子就像一个巨大的幼儿园，所有一切都是为了教育下一代而有意识地造出来的。

在我生长的这个地方，感觉整个世界就是这么大，你根本没有别的地方可去，没有别的事情可做。这一点都不令人意外。郊区就是故意这样设计的，与外部世界隔离，不让儿童沾染到外界有害的东西。

至于学校，不过是这个虚假环境中关住牲口的围栏。表面上，学校的使命是教育儿童。事实上，学校的真正目的是把儿童都关在同一个地方，以便大人们白天可以腾出手来把事情做完。我对这一点没有意见，

在一个高度工业化的社会，对孩子不加管束，让他们四处乱跑，无疑是一场灾难。

让我困扰的，不是把孩子关在监狱里，而是（a）不告诉他们这一点，（b）把这监狱的大部分交给犯人来管理。孩子们被送进来，花 6 年时间，记住一些毫无意义的事实，还要身处在一个由四肢发达的小巨人管理的世界，那些巨人们只知道追逐一个椭圆形的、棕色的球[①]，好像这是全世界最天经地义的事情。这简直就像一场超现实的鸡尾酒化妆晚会，如果孩子畏缩不前、瑟瑟发抖，他们就会被视为怪人。

生活在这个扭曲的世界，不仅仅对书呆子，对所有孩子来说，都是充满压力的。就像任何一场战争，胜利方也是要付出代价的。

成年人肯定不可避免地看到了孩子们在受苦受难。他们为什么不做点什么呢？因为他们认为那是青春期在作祟。成年人对自己说，孩子们不快乐的原因是因为他们身体内部新出现了大量的化学物质——激素。激素在血液中奔流，把所有事情都搞得一团糟。整个社会系统一点问题也没有，孩子们到了这个年纪，不可避免地会感觉很糟糕。

这种看法无所不在，甚至孩子们自己都相信了。但是相信这种话可能一点帮助也没有。你告诉一个人，他的脚天生就是坏的，并不能阻止他去怀疑他可能穿错了鞋子。

我就不太相信这种理论，凭什么说 13 岁的小孩自己有问题。如果这是激素过多的生理问题，那就应该普遍存在。可是，蒙古的游牧民族在 13 岁时难道也是这么空虚吗？我读过许多历史资料，找不到任何一条 20 世纪之前的历史事实支持这个理论上应该普遍存在的现象。文艺复兴时期的很多青少年学徒看上去过得很开心很投入。当然，他们彼此之间也有争斗和阴谋诡计（米开朗基罗小时候就曾经被其他小孩打断过鼻子），但是他们并不疯狂。

就我所知，青少年因为激素而行为失常的理论与美国中产阶级迁至郊区的进程是同步出现的。我认为这不是巧合，青少年是被迫去过这种

① 指橄榄球。——译者注



生活的，他们是被逼疯的。文艺复兴时期的学徒是整天劳碌的牧羊狗，而今天的青少年则是神经兮兮、供人玩耍的哈巴狗。他们的疯狂源于到处都是一片可怕的无聊。

我读中学的时候，自杀是聪明学生中永恒的话题。虽然在我认识的人当中没有人真的自杀，但是不少人有这样的设想，其中一些人可能还真的尝试过。对于大多数人来说，自杀只是一个姿态。就像其他青少年一样，我们都喜欢有戏剧效果，而自杀看上去就非常富有戏剧性。但是也有部分原因是因为我们的生活有时真的是非常悲惨。

被其他小孩欺负只是问题的一部分。还有别的问题存在，甚至可能是更糟糕的问题。那就是我们没有得到真正的工作，没能发挥我们的才能。人类喜欢工作，在世界上大多数地方，你的工作就是你的身份证明。但是，我们那时做的所有事情根本就是无意义的，至少那时看来是这样。



最好的情况下，那些事情也不过是遥远的将来我们可能从事的实际工作的练习。它所面向的目标是如此遥远，以至于当时我们都不知道自己练习这些到底是为了干什么。更常见的情况是，那些事情不过是一系列随意设置的绳圈，你被要求一个个跳过去。你在学习中遇到的文字都是专为考试而设计的，目的就是为了出题，而不是为了讲清楚问题。（南北战争的三个主要原因是……等到考试的时候，就会有一道题：请列出南北战争的三个主要原因。）

而且，没有办法回避那些事情。成年人已经达成共识，认定通往大学的途径就是这样的。逃离这种空虚生活的唯一方法，就是向它屈服。

过去的社会中，青少年扮演着一个更积极的角色。工业化时代到来前，青少年都是某种形式的学徒，不是在某个作坊，就是在某个农庄，甚至在某艘军舰上。他们不会被扔到一旁，创造自己的小社会。他们是成年人社会的低级成员。

以前的青少年似乎也更尊敬成年人，因为成年人都是看得见的专家，会传授他们所要学习的技能。如今的大多数青少年，对他们的家

长在遥远的办公室所从事的工作几乎一无所知。他们看不到学校作业与未来走上社会后从事的工作有何联系（实际上，还是有那么一点点联系）。

如果青少年更尊重成年人，那么成年人也会更接受青少年。经过几年的训练，学徒就能担当重要的职责。即使是那些刚招收进来的学徒，也能用来送信或打扫场地。

如今的成年人根本不接受青少年。一般来说，他们都是在办公室工作，所以就在上班的路上，顺路把孩子送到学校去关着，这有点像他们周末外出度假时，把狗送到寄养的地方。

与此同时，社会发生了什么变化？我们被迫面对一个更严峻的问题。它与当前的其他许多难题有着共同的起因，那就是“专业化”(specialization)。当工作的专业程度越来越高时，我们就必须接受更长时间的训练。工业化时代来临之前，儿童最晚大约在 14 岁就要参加工作，如果是在农庄(那个时代大多数人生活在农村),参加工作的时间就更早。如今，只要一个青少年读大学，他就要等到 21 岁或 22 岁才开始全职工作。如果再读更高的学位，比如医学博士或哲学博士，可能要拖到 30 岁才能完成学业。

当今的青少年在生产活动中，根本就是毫无用处的。他们只能在诸如快餐店这样的地方充当廉价劳动力，而快餐店也看出来了，充分利用了这个事实。对于除此以外的几乎所有行业，青少年都会带来净损失。但是，他们又太年轻，不能放任不管，必须有人看着他们。最有效的解决方案，就是把他们集中在一个地方，用几个成年人看守所有小孩。

如果事情只发展到这一步，那么我们就是在描述一个监狱，唯一的区别就是这个监狱不是全日制的。问题在于，许多学校实际上真的停留在这一步。学校的使命据称是教育儿童，但是并没有外在的压力监督他们把这件事做好。所以，大多数学校的教学质量都很糟糕，孩子们根本不把学习当回事，就连认真读书的孩子也是如此。许多时候，我们所有人——包括学生和老师——都只是做做样子，走过场而已。

我在高中上法语课的时候，课程内容包括阅读雨果的长篇小说《悲惨世界》。我觉得，学生中不可能有人的法语水平高到可以自己读懂这本



巨著。所以，像班上的其他人一样，我参考了 *Cliff's Notes*[①]的导读本。后来，学校有一次专门针对《悲惨世界》的测验，我发现里面的问题都很奇怪，到处都是很长的单词，老师上课时从没有用过这些词。这些题目是从哪里来的？原来也是出自 *Cliff's Notes*。老师们也在使用这个导读本，我们双方都是在敷衍了事。

公立学校肯定也有很优秀的老师。我读四年级时遇到的 Mihalko 老师就是一个精力充沛、充满想象力的老师。他使得那个学年如此令人难忘，以至于三十年后，他的学生依然在谈论这段往事。但是，像他这样的老师只是个别现象，无法改变整个体系。

几乎在任何团体中都存在等级关系。成年人在真实世界中形成的团体，一般来说，都存在某个共同目标，团体的领导者往往由最善于实现目标的人担任。学校就不一样，大多数情况下，学生内部形成的团体没有一个共同目标。但是，等级关系却不会缺席，所以孩子们的等级是凭空创造出来的。

我们有一个专门的短语描述这种情况，即在没有任何严肃标准的前提下，产生排名的情况。我们会说情况“倒退至人缘比赛”（degenerates into a popularity contest）。这正是大多数美国学校中发生的事情。某个人的排名不是根据他的真正能力，而主要根据他专攻排名的能力。这就像路易十四的宫廷[②]。没有外在的对手，孩子们就互相把对方当作对手。

如果存在对于真正能力的外部测试，待在等级关系的底层也不会那么痛苦。球队的新人并不会怨恨老队员的球技，他希望有一天自己也能如此，所以很高兴有机会向老队员求教。老队员可能也会因此产生一种传帮带的光荣感（noblesse oblige）。最重要的是，老队员的地位是通过他们本身出色的能力获得的，而不是通过排挤他人获得的。

宫廷中的等级关系就完全是另一回事了。这种类型的团体贬低了每

① *Cliff's Notes*是美国的一套系列图书，专供学生作为课外的学习参考读物。——译者注

② 路易十四（Louis XIV，1638—1715）是法国国王，1643年~1715年在位，欧洲君主专制制度的典型代表。他大权独揽，将各地的贵族都召集在他的宫廷中，削弱他们作为地方长官的权利。贵族们为了得到路易十四的赏识，不得不每天精心准备，参加各种凡尔赛宫的舞会、宴席和其他庆祝活动，想尽方法，互相竞赛，试图压过对手。——译者注



一个成员的人格。底层成员对上层成员毫无敬意，而上层成员也没有传帮带的光荣感。这里的一切就是杀与被杀的关系。

美国中学的学生内部，就是这种关系的社会。因为除了每天把小孩聚集在某个地方，关上几个小时以外，学校并没有其他的真实目的，所以学生内部形成这种关系也就很自然了。我当时并没有意识到，直到不久以前才恍然大悟，校园生活的两大恐怖之处——残忍和无聊——也是出于同样的原因。

美国公立学校的平庸并不仅仅是让学生度过了不快乐的六年，还带来了严重后果。这种平庸直接导致学生的叛逆心理，使他们远离那些原本应该要学习的东西。

许多书呆子可能都与我一样，直到高中毕业多年后，才去读中学里的指定读物。但是，我错过的绝不仅仅只是几本书而已。我对许多美好的字眼都嗤之以鼻，比如“人格”、“正直”，因为成年人贬低了这些词。在他们嘴里，这些词似乎都是同一个意思——“听话”。一些孩子因为具备所谓的“人格”和“正直”，而受到夸奖，可是他们不是呆得像一头大笨牛，就是轻浮得像一个不动脑筋的吹牛者。如果“人格”和“正直”就是这种样子，我宁愿不要它们。

我误解最深的一个词是“老成”（tact）。成年人使用这个词，含义似乎就是“闭上嘴巴，不要说话”。我以为它与“缄默”（tacit）和“不苟言笑”（taciturn）有着相同的词根，字面意思就是安静。我就对自己发誓，我绝不要变成“老成”的人，没有人能够让我闭上嘴巴。可是事实上，这个词的词根与“触觉”（tactile）相同，它真正的意思是熟练的碰触。“老成”的反义词是“笨拙”（clumsy）。进入大学以后，我才搞明白了这个词。

在这场激烈的人缘争斗中，书呆子并不是唯一输家。他们不受欢迎，只是因为他们分心去干别的事了。还有一些孩子则是主动放弃，因为他们对这个过程感到厌恶。

青少年都不喜欢孤独一人，即使具有叛逆心理的青少年也是如此。所以，当他们选择退出这个系统时，他们往往会一群人一起退出。在我



读过的学校，叛逆心理的主要焦点在于毒品的使用，尤其是大麻。这帮孩子穿着黑色的演唱会 T 恤，被称为“怪人”（freak）①。

怪人和书呆子属于同一个联盟，他们之间有很多相同之处。虽然在怪人的群体中，绝不学习是一个很重要的价值观（至少看上去如此），但是从整体上看，怪人还是比其他孩子聪明。我本人更多地属于书呆子阵营，但是我有很多怪人朋友。

怪人使用毒品，是为了建立他们之间的社交纽带，至少一开始如此。因为毒品是非法的，所以一起使用的话，就创造出了一种共同反叛的标志。

我在这里并不是说糟糕的学校教育是孩子使用毒品的唯一原因。只要你碰过毒品，一段时间以后，毒品就会自行推动你的行为。而且毫无疑问，有些怪人使用毒品的根本原因是为了逃避其他问题，比如家庭问题。但是，至少在我的学校，大部分孩子使用毒品的主要原因是出于叛逆心理。14 岁的孩子开始抽大麻，并不是因为他们听说这样有助于忘记烦恼，而是因为他们想要加入一个不同的团体。

不当的管制导致叛逆，这并不是新鲜事。即使毒品本身就是麻烦的来源，学校本身依然无法推卸主要责任。



校园生活的真正问题是空虚。除非成年人意识到这一点，否则无法解决这个问题。可能意识到这个问题的成年人，是那些读书时就是书呆子的人。你想让你的孩子读八年级的时候和你一样不快乐吗？我可不想。那好，有什么事是我们可以做的？肯定有的。现行体系中没有什么事是必然的。它是现在这个样子，大部分是因为没人去改变它②。

你也许会说，成年人很忙。观看孩子在学校的文艺表演是一回事，

① 除了“行为怪异”以外，freak还指通过吸毒逃避现实的人，以及吸毒引起的幻觉。——译者注

② 那么我们应该怎么改变中学？一个可能的答案就是参考大学的样子。如果你进入了一所（好的）大学，我提到的大部分问题就都得到了解决。所以，解决方案也许就是问，你怎样才能让青少年书呆子的生活，变得更像大学生活？不让孩子上学，进行家庭教育，也是一个解决方法，而且很直接。但是，它可能并不是最好的方法。否则的话，家长为什么不让孩子一直待在家里，直到把大学也教完了呢？因为大学能够提供家庭教育无法提供的东西。同样的道理，如果处理得当，中学也能够提供家庭教育无法提供的东西。

着手改革教育制度又是另一回事。可能只有少数人有精力从事改革工作。那么我要说，我觉得最难的部分，其实是能否意识到你该做什么。

还在学校里读书的书呆子不应该屏息凝神，等着全副武装的成年人某一天乘直升飞机从天而降来拯救你。也许会有这么一天，但是肯定不会很快到来。任何对生活立竿见影的改变，可能还是来自于书呆子自己。

哪怕你什么也改变不了，但是仅仅是理解自己的处境，也能使得痛苦减轻一些。书呆子并不是失败者。他们只是在玩一个不同的游戏，一个更接近于真实世界状况的游戏。成年人明白这一点。成功的成年人，几乎都声称自己在高中属于书呆子。

对于书呆子来说，意识到学校并非全部的人生，也是很重要的事情。学校是一个很奇怪的、人为设计出来的体系，一半像是无菌室，一半像是野蛮洪荒之地。它就像人生一样，里面无所不包，但又不是事物的真实样子。它只是一个暂时的过程，只要你向前看，你就能超越它，哪怕现在你还是身处其中。



如果你觉得人生糟透了，那不是因为体内激素分泌失调（你父母相信这种说法），也不是因为人生真的糟透了（你本人相信这种说法）。那是因为你对成年人不再具有经济价值（与工业社会以前的时期相比），所以他们把你扔在学校里，一关就是好几年，根本没有真正的事情可做。任何这种类型的组织都是可怕的生存环境。你根本不需要寻找其他的原因就能解释为什么青少年是不快乐的。

我在这篇文章中发表了一些刺耳的意见，但是我对未来是乐观的。我们认定无法解决的难题，事实上完全可以解决。青少年并不是洪水猛兽，也并非天生就不快乐。这一点对于青少年和成年人，应该都是令人鼓舞的消息。



2 黑客与画家

读完计算机系的研究生，我就去了艺术学校，学习绘画。许多人很吃惊：一个喜欢计算机的人，居然还喜欢画画！他们似乎觉得，摆弄计算机和画画是两件截然不同的事情——计算机是冰冷的、精确的、井然有序的，而画画是某种原始欲望热烈狂放的表达方式。

这种看法是错的。计算机和画画有许多共同之处。事实上，在我知道的所有行业中，黑客与画家最相像。



黑客与画家的共同之处，在于他们都是创作者。与作曲家、建筑师、作家一样，黑客和画家都是试图创作出优秀的作品。他们本质上都不是在做研究，虽然在创作过程中，他们可能会发现一些新技术（那样当然更好）。

我一直不喜欢“计算机科学”（computer science）这个词。主要原因是根本不存在这种东西。计算机科学就像一个大杂烩，由于某些历史意外，很多不相干的领域被强行拼装在一起。这个学科的一端是纯粹的数学家，他们自称“计算机科学家”，只是为了得到国防部研究局（DARPA）的项目资助。中间部分是计算机博物学家，研究各种专门性的题目，比如网络数据的路由算法。另一端则是黑客，只想写出有趣的软件，对于他们来说，计算机只是一种表达的媒介，就像建筑师手里的混凝土，或者画家手里的颜料。所以，在“计算机科学”的名下，数学家、物理学家、建筑师都不得不待在同一个系里。

有时，黑客做的事情被称为“软件工程”（software engineering），但是这个词也是误导的。与其说优秀的软件设计师是工程师，还不如说是

建筑师[1]。建筑学和工程学之间的区别并不是很严格的，但就是存在区别。这表现在“做什么”和“怎么做”：建筑师决定做什么，工程师想出怎么做。

当然，“做什么”和“怎么做”不应该分得太开。如果你决定做一件事，却不知道怎么做，你就是在自找麻烦。但是，只是单纯地决定如何实现某种规格，那肯定不是黑客。黑客的最高境界是创造规格。虽然看起来，做到这一点的最好方法就是先做出一个样品把规格实现了。

也许有一天，“计算机科学”分裂成几个独立的部分。这可能是一件好事，如果我本人的领域——黑客——能够独立出来，那就更好了。

把不同类型的工作捆绑在一起，可能是为了行政管理的方便，但是却容易引起混淆。这是我不喜欢“计算机科学”这个词的又一个原因。中间部分“计算机科学家”的工作，也许还可以被称为计算机的实验科学。但是，两端的数学家和黑客，并不是在做计算机的科学研究。



数学家看来并不在乎自己搞的是计算机还是数学。他们很高兴来到这个新地方，然后就开始埋头证明新的定理，与数学系的数学家干的事情完全一样。不一会儿，他们可能就忘了办公楼外的牌子上写的是“计算机科学系”。但是对于黑客，“计算机科学”这个标签是一个麻烦。如果黑客的工作被称为科学，这会让他们感到自己应该做得像搞科学一样。所以，大学和实验室里的黑客，就不去做那些真正想做的事情（设计优美的软件），而是觉得自己应该写一些研究性的论文。

要是黑客写论文，最好的情况下，写出来的也只是一些补充性的描述，不会具有太大的实际价值。黑客先开发了一个很酷的软件，然后就写一篇论文，介绍这个软件。论文变成了软件成果的展示。这种结合是错误的，常常会产生问题。为了配合论文研究性的主题，你很容易就把工作重点从开发优美的软件转移到开发一些丑陋的东西。

优美的软件并不总是论文的合适题材。首先，科学研究必须具有原

① 在英语中，“建筑师”（architect）和“架构师”（architect）是同一个词，所以这里用的是双关语，意思是优秀程序员不仅负责建造，还负责架构。后一句中的“建筑学”（architecture）也是这种双关用法，同时指“架构学”（architecture）。——译者注

创性。写过博士论文的人都知道，确保自己正在开垦新领地的方法，就是去找那些没有人要的土地。其次，科学研究必须是能够产生大量成果的，而那些不成熟的、障碍重重的领域最容易写出许多篇论文，因为你可以写那些为了完成工作、你不得不克服的障碍。没有什么比一个错误的前提更容易产生大量待解决的问题了。人工智能（AI）领域的大部分情况，都符合这条“如何凭空创造出问题”的规律。如果你假定，使用一系列的谓词逻辑（predicate logic）表达式，再加上代表抽象概念的参数，就能表达人类的知识，那么，你就可以写出许许多多的论文，解释如何完成这项工作。这就像电视剧《我爱露西》（*I Love Lucy*）的男主角 Ricky Ricardo 的话：“露西，这下够你好好解释的了。”

创造优美事物的方式往往不是从头做起，而是在现有成果的基础上做一些小小的调整，或者将已有的观点用比较新的方式组合起来。这种类型的工作很难用研究性的论文表达。



那么，为什么大学和实验室还把论文数量作为考核黑客工作的指标呢？这种事情其实在日常生活中普遍存在，比如，我们使用简单的标准化测试考核学生的“学术能力倾向”（scholastic aptitude），再比如，我们使用代码的行数考核程序员的工作效率。这样的考核容易实施，而容易实施的考核总是首先被采用。

黑客真正想做的是设计优美的软件，考核这种工作是非常困难的。你本人需要有良好的设计感，才能去考核别人的设计是否良好。但是，你觉得你有“良好的设计感”，与你实际是否具有，不存在相关关系，甚至可能存在负相关。

唯一有效的外部考核就是时间。经过岁月的洗礼，优美的东西生存发展的机会更大，丑陋的东西往往会被淘汰。不幸的是，这种考核需要的时间可能比一个人的生命还要长。塞缪尔 • 约翰逊[①]说过，人们对一

---

① 塞缪尔 • 约翰逊（Samuel Johnson，1709—1784），英国词典学家，编撰了历史上第一本广泛使用、影响巨大的英文字典。——译者注

个作家的评价，需要 100 年才能达成一致[①]。你必须先等他的那些有影响力的朋友都死了，然后再等他的追随者都死了，才能对他有一个公正的评价。

我想，名望有很大的随机性，黑客对此只好听天由命了。在这一点上，他们与其他创作者并无不同。事实上，相比而言，他们还是幸运的。暂时性的、一窝蜂式的时代风潮对画家的影响要比对黑客的影响大得多。

人们无法考核你的工作，甚至误解你的工作，都不是最糟的事。更大的危险是你自己也会误解自己的工作。因为你总是从相关领域寻找新思想，如果你发现自己读的是计算机科学系，很自然地，你就会以为“计算机科学”与其他“理论科学”并无不同，你的工作属于“理论计算机科学”所涉及的那种理论的应用研究。读研究生期间，我潜意识里一直有一种很不舒服的感觉，觉得自己应该多学一点理论，不应该期末考试结束还不到三个星期，就把所有东西忘得一干二净，那样真是不可饶恕。



现在，我意识到自己错了。黑客搞懂“计算理论”（theory of computation）的必要性，与画家搞懂颜料化学成分的必要性差不多大。一般来说，在理论上，你需要知道如何计算“时间复杂度”和“空间复杂度”（time and space complexity）；如果你要写一个解析器，可能还需要知道状态机（state machine）的概念；除此以外，并不需要知道特别多的理论。这些可比画家必须记住的颜料成分少很多。

我发现，黑客新想法的最佳来源，并非那些名字里有“计算机”三个字的理论领域，而是来自于其他创作领域。与其到“计算理论”领域寻找创意，你还不如在绘画中寻找创意。

---

① 塞缪尔·约翰逊在他编辑的《莎士比亚戏剧集》的前言中写道：“他（莎士比亚）的影响远远超过他的时代，时间就是对他文学成就的检验。不管他的作品从那时的暗语、风俗、政治局势之中，得到过怎样的优势，这些优势都已经消逝多年了。他在每一幕戏剧中，通过模拟那时的生活所产生的每一个欢乐的主题或悲伤的动机，都已经趋于平淡，而不再是戏剧的亮点。贵族的宠爱和对手的竞争，都不再产生效果；朋友和敌人都走进了坟墓；他的作品再也不是支持一方、打击另一方的舆论工具；它们既不能产生虚名，也不会带来恶意的攻击。人们阅读这些作品，只有一个理由，那就是欣赏作品本身。因此，只有人们真正欣赏它们，才会发出赞美……”

举例来说，我在大学受到的教育是，在上机编程之前，应该先在纸上把程序搞清楚。可我自己一直不是这样编程的，我喜欢直接坐在计算机前编程，而不是在纸上编程。更糟的是，我不是耐心地一步步写出整个程序，确保大体上是正确的，而是一股脑不管对错，先把代码堆上去，再慢慢修改。书上说，调试（debugging）是最后的步骤，用来纠正打字的错误和疏忽。可是我的工作方法看上去却像编程就是在调试。

很长一段时间内我都为此事沮丧，就像小学里老师教我怎么拿铅笔，我却总是学不会的那种感觉。如果我那时看到其他创作领域，比如绘画或者建筑，我就会想到，自己的方法其实有一个正式的名称：打草稿。我现在认为，大学里教给我的编程方法都是错的。你把整个程序想清楚的时间点，应该是在编写代码的同时，而不是在编写代码之前，这与作家、画家和建筑师的做法完全一样。



明白这一点对软件设计有重大影响。它意味着，编程语言首要的特性应该是允许动态扩展（malleable）。编程语言是用来帮助思考程序的，而不是用来表达你已经想好的程序。它应该是一支铅笔，而不是一支钢笔。如果大家都像学校教的那样编程，那么静态类型①（static typing）是一个不错的概念。但是，我认识的黑客，没有一个人喜欢用静态类型语言编程。我们需要的是一种可以随意涂抹、擦擦改改的语言，我们不想正襟危坐，把一个盛满各种变量类型的茶杯，小心翼翼放在自己的膝盖上，为了与一丝不苟的编译器大婶交谈，努力地挑选词语，确保变量类型匹配，好让自己显得礼貌又周到。

创作者不同于科学家，明白这一点有很多好处。除了不用为静态类型烦恼以外，还可以免去另一个折磨科学家的难题，那就是“对数学家的妒忌”。科学界的每一个人，暗地里都相信数学家比自己聪明。我觉得，数学家自己也相信这一点。最后的结果就是科学家往往会把自己的工作尽可

① 静态类型是某些计算机语言的一个特性，指编译时对变量类型进行严格检查，典型代表是C、C++和Java。在这一类语言中，声明变量的时候，必须指定类型，而且以后不能再改变。这必然意味着，只有在你对整个程序流程和细节思考成熟以后，才能编写代码。与之对应的则是动态类型（dynamic typing）语言，变量包含的数据类型可以随时改变。——译者注

能弄得看上去像数学。对于物理学这样的领域，这可能不会有太大不良影响。但是，你越往自然科学的方向发展，它就越成为一个严重的问题。

一页写满了数学公式的纸真是令人印象深刻啊。（小窍门：用希腊字母表示变量名会令人印象更深刻。）因此，你就受到巨大的诱惑，去解决那些能够用数学公式处理的问题，而不是去解决真正重要的问题。

如果黑客认识到自己与其他创作者——比如作家和画家——是一类人，这种诱惑对他就不起作用。作家和画家没有“对数学家的妒忌”，他们认为自己在从事与数学完全不相关的事情。我认为，黑客也是如此。

如果大学和实验室不允许黑客做他们想做的事情，那么适合黑客的地方可能就是企业。不幸的是，大多数企业也不允许黑客做他们想做的事情。大学和实验室强迫黑客成为科学家，企业强迫黑客成为工程师。

直到最近我才发现这一点。雅虎收购 Viaweb[①]的时候，他们问我想做什么。我对商业活动从来都没有太大兴趣，就回答说我想继续做黑客。等我来到雅虎以后，发现在他们看来，“黑客”的工作就是用软件实现某个功能，而不是设计软件。在那里，程序员被当作技工，职责就是将产品经理的“构想”（如果这个词是这么用的话）翻译成代码。



这似乎是大公司的普遍情况。大公司这样安排的原因是为了减少结果的标准差。因为实际上只有很少一部分黑客懂得如何正确设计软件，公司的管理层很难正确识别到底应该把设计软件的任务交给谁。所以，大部分公司不把设计软件的职责交给一个优秀的黑客，而是交给一个委员会，黑客的作用仅仅是实现那个委员会的设计。

如果某一天你想要去赚大钱，那么记住上面这一点，因为这是创业公司能够成功的原因之一。大公司为了避免设计上的灾难，选择了减少设计结果的标准差。但是当你排斥差异的时候，你不仅将失败的可能性排除在外，也将获得高利润的可能性排除在外。这对大公司来说不是问题，因为生产特别优秀的产品不是它们的获胜手段。大公司只要做到不

---

① Viaweb是一个帮助用户开设网上商店的互联网应用程序，被认为是第一个互联网应用程序，由作者和罗伯特·莫里斯在1995年创立。1998年，雅虎以4900万美元购买了Viaweb，更名为Yahoo! Store。——译者注

太烂，就能赢。

所以，如果你的竞争优势是在软件设计方面，并且你的对手是一家大公司，它大到由一群产品经理来设计软件，那么你的对手将永远无法赶上你。不过说实话，这样的机会不容易找到。你很难单单依靠软件设计就与大公司展开竞争。这就好比你很难攻入城堡与对手面对面地徒手搏斗。比如，就算写一个比微软的 Word 更好的文字处理软件不是难事，但是微软公司有自己的城堡，它的操作系统是垄断的，你根本无法对它构成威胁，它甚至都不会注意到你的存在。

真正竞争软件设计的战场是新兴领域的市场，这里还没有人建立过防御工事。只要你能做出大胆的设计，由一个人或一批人同时负责设计和实现产品，你就能在这里战胜大公司。微软公司自己一开始就是这样走向成功的，苹果公司和惠普公司也是如此。我觉得几乎所有的创业公司都是这样取得成功的。



所以，开发优秀软件的方法之一就是自己创业。但是，这样做会遇到两个问题。一个是自己开公司的话，必须处理许许多多与开发软件完全无关的事情。我创立 Viaweb 的时候，如果有四分之一时间可以用于开发，就感到很幸运了。我在其他四分之三时间所做的事情，从单调乏味到惊悚恐怖，无所不包。我来说一个比较，有一次董事会开到一半，我不得不离开去补牙。我记得坐在牙医诊所的椅子上，等着医生开动牙钻的那段时间，与刚才待在公司的时间相比，简直感觉像是在度假一样。

创业的另一个问题是赚钱的软件往往不是好玩的软件，两者的重叠度不高。设计编程语言是很好玩的事情，事实上，微软的第一个产品就是一种编程语言[①]。但是，如今没有人会出钱买编程语言。如果你想赚钱，你可能不得不去干那些很麻烦很讨厌的事情，因为这些事情没人愿意义务来干。

所有创作者都面临这个问题。价格是由供给和需求共同决定的。好玩的软件的需求量，比不上解决客户麻烦问题的软件的需求量。在小剧

---

① 1975年，MITS公司发布了Altair 8800型计算机，比尔· 盖茨意识到为它开发软件是有利可图的，他写了一个BASIC语言解释器，卖给了MITS。这就是微软公司的第一个产品，当时比尔·盖茨还是哈佛大学二年级在校生。——译者注

场里演出的酬劳，比不上穿着卡通大猩猩服装、在展览会上为厂商站台的酬劳。写小说的回报比不上写广告文案的回报。开发编程语言的收入，比不上把某些公司老掉牙的数据库连上服务器的收入。

黑客如何才能做自己喜欢的事情？我认为这个问题的解决方法是一个几乎所有创作者都知道的方法：找一份养家糊口的“白天工作”（day job）。这个词是从音乐家身上来的，他们晚上表演音乐，所以白天可以找一份其他工作。更一般地说，“白天工作”的意思是，你有一份为了赚钱的工作，还有一份为了爱好的工作。

几乎所有的创作者在职业生涯的早期都有一份“白天工作”。画家和作家尤其显著。如果幸运的话，你能找到一份与你的“真正工作”非常相关的“白天工作”。音乐家似乎常常是在唱片行工作。同样地，钻研某种编程语言或操作系统的黑客，很可能会得到一份使用这些工具的“白天工作”。①



当我说，黑客解决生计问题的方法是找一份“白天工作”，然后在其余时间开发优美的软件，我并没有说这是一个新方法。开源软件界的黑客早就这样做了。我想说的其实是，开源软件的这种工作模式可能就是正确的模式，因为它已经被其他领域的创作者都验证过了。

令我惊讶的是，雇主都很犹豫，不愿意手下的黑客为开源软件项目工作。但是，在 Viaweb，要是你不愿意这样干，我们会很犹豫要不要雇用你。我们面试程序员的时候，主要关注的事情就是业余时间他们写了什么软件。因为如果你不爱一件事，你不可能把它做得真正优秀，要是你很热爱编程，你就不可避免地会开发你自己的项目。②

---

① 摄影技术对绘画造成的最大伤害，也许就是消灭了画家最好的“白天工作”。历史上，大多数伟大画家都靠画肖像谋生。摄影术发明不久，画肖像的工作机会就大大减少，从事摄影的黑客抢走了这些机会。（对于被画的人来说，坐在镜头前也更轻松一些。）肖像画家——这种要求高度技巧的工作——就这样或多或少地消失了。画家为了得到收入，就将肖像技巧用在为商业公司画品牌商品的广告图片。（可是，这样的工作也是极大地依赖于摄影术，或者更准确地说，极大地依赖于复制在书籍和杂志上的照片。）

② 微软不鼓励雇员为开源项目做贡献，甚至业余时间也不行。但是，如此之多的一流黑客都在从事开源项目，所以这个政策主要的效果，可能就是使得微软公司很难雇到一流的程序员。

因为黑客更像创作者，而不是科学家，所以要了解黑客，不应该在科学家身上寻找启示，而是应该观察其他类型的创作者。那么，从画家身上，我们还能借鉴到什么对黑客的启示呢？

有一件事情是可以借鉴的（至少可以确认），那就是应该如何学习编程。画家学习绘画的方法主要是动手去画，黑客学习编程的方法也理应如此。大多数黑客不是通过大学课程学会编程的，他们从实践中学习，13 岁时就自己动手写程序了。即使上了大学，黑客学习编程依然主要通过自己写程序。①

画家的作品都会保留下来，你观察这些作品，就能看出他们是怎么一步步通过实践学习绘画的。如果你把一个画家的作品按照时间顺序排列，就会发现每幅画所用的技巧，都是建立在上一幅作品学到的东西之上。某幅作品如果有特别出色之处，你往往能够在更早的作品上发现一个小规模的初期版本。



我想大多数创作者都是这样学习和工作的，作家和建筑师似乎都是如此。也许对于黑客来说，采取像画家这样的做法很有好处：应该定期地从头开始，而不要长年累月地在一个项目上不断工作，并且试图把所有的最新想法都以修订版的形式包括进去。

黑客通过实践学习编程，这又是一个标志，说明黑客与科学家的区别有多大。科学家就不会通过干活来学习科学，而是通过做实验和解题来学习。科学家研究的基础都是现有的很完美的成果，在这个意义上，他们的第一步只是在复制别人已经做过的工作。最后，他们才会从某一个点开始，进行自己的原创性工作。但是，黑客就不一样，从一开始做的就是原创性工作，根本没有他人完美的成果可以依靠。所以，黑客的出发点是原创，最终得到一个优美的结果；而科学家的出发点是别人优美的结果，最终得到原创性。

创作者另一个学习的途径是通过范例。对画家来说，博物馆就是美术技巧的图书馆。几百年来，临摹大师的作品一直是传统美术教育的一部分，因为临摹迫使你仔细观察一幅画是如何完成的。

---

① 大学里学习编程，就像读书和挑选衣服一样，你会发现自己高中时的品味是多么糟糕。

作家也是这样学习写作的。富兰克林（Benjamin Franklin）通过总结和模仿艾迪生和斯梯尔[①]的文章，学会了写作。雷蒙·钱德勒[②]（Raymond Chandler）也是如此学会了写作侦探小说。

同样地，黑客可以通过观看优秀的程序学会编程，不是看它们的执行结果，而是看它们的源代码。开源运动最鲜为人知的优点之一，就是使得学习编程变得更容易了。我学编程的时候，不得不主要依靠教材上的范例。那时可以搞到的源码，主要来自于 Unix，但是就连 Unix 也不是开源的。大部分阅读 Unix 源码的人都是通过约翰·莱昂斯[③]那本书的非法影印本。该书虽然是 1977 年写的，但是在 1996 年之前都不被允许公开出版。

还有一个可以借鉴绘画的地方：一幅画是逐步完成的。通常一开始是一张草图，然后再逐步填入细节。但是，它又不单纯是一个填入细节的过程。有时，原先的构想看来是错的，你就必须动手修改。无数古代油画放在 X 光下检视，就能看出修改痕迹，四肢的位置被移动过，或者脸部的表情经过了调整。



绘画的这个创作过程就值得学习。我认为黑客也应该这样工作。你不能盼望先有一个完美的规格设计，然后再动手编程，这样想是不现实的。如果你预先承认规格设计是不完美的，在编程的时候，就可以根据需要当场修改规格，最终会有一个更好的结果。

（大公司的内部结构，使得它们很难这样做。这是又一个创业公司占优之处。）

眼下想必每个人都知道，过早优化（premature optimization）是一件

---

① 此处的艾迪生指Joseph Addison（1672—1719），斯梯尔指Richard Steele（1672—1729），两人都是18世纪初的英国作家，于1711年共同创办了政论讽刺杂志《旁观者》（*The Spectator*），产生了巨大的影响。——译者注

② 雷蒙·钱德勒（1888—1959），美国推理小说作家，他的私人侦探菲力普·马罗（Philip Marlowe）系列小说有很大的读者群。——译者注

③ John Lions（1937—1998），澳大利亚计算机科学家。1976~1977年为了授课需要，他写了《UNIX第6版源码注释》（*Lions' Commentary on UNIX 6th Edition, with Source Code*）一书。很长一段时间中，该书是贝尔实验室之外唯一的UNIX内核源码文档。由于UNIX第6版源码只允许用于教学，所以该书直到1996年才公开出版，此前一直都是私下传播。人们普遍相信它是计算机科学领域被复印次数最多的书。——译者注

危险的事情。我认为，我们应该对“过早设计”（premature design）也抱有同样的担忧，不要太早决定一个程序应该怎么做。

合适的工具能够帮助我们避免这种危险。一种好的编程语言，应该像油画颜料一样，能够使得我们很从容地改变想法。动态类型语言在这一点上就是赢家，因为你不必提前就设置好各种变量的数据类型。不过我认为，编程语言灵活性的关键还不在这里，而在于这种语言应该非常抽象。最容易修改的语言就是简短的语言。

我接下来要说的是，一幅优秀的绘画作品必须比它应该有的样子更好，这可能听起来有点矛盾。举例来说，图 2-1 所示达·芬奇的作品《女性肖像》（*Ginevra de' Benci*）现在保存在美国国家美术馆。你可以看到，达·芬奇在少女的头后面摆了一片树枝。他很仔细地画出了树枝上的每一片叶子。许多画家也许会觉得，那不过是放在背景里的衬托物，没有人会仔细看的，不妨简单处理一下就可以了。





图 2-1　达·芬奇的《女性肖像》（*Ginevra de' Benci*），1474 年

但是达·芬奇不这样想。他对作品每一部分的认真程度完全不取决于预料中会不会有人仔细看这个部分。他就像篮球巨星迈克尔·乔丹

（Michael Jordan），每一球都一丝不苟，绝不降低对自己的要求。

坚持一丝不苟，就能取得优秀的成果。因为那些看不见的细节累加起来，就变得可见了。当人们从达·芬奇的《女性肖像》前面走过的时候，他们的注意力往往立刻就被它吸引住了，那时他们甚至还没有看到说明的标签牌，没有意识到这是达·芬奇的作品。所有那些看不见的细节合并在一起，就使得这样东西产生了惊人的效果，仿佛上千个细微的声音都以同一个音调在歌唱。

同样地，优秀的软件也要求对美的狂热追求。如果你查看优秀软件的内部，就会发现那些预料中没有人会看见的部分也是优美的。我对待代码的认真程度远远超过我对待其他事情，如果我以这种态度对待日常生活的每件事，那么我就够资格找心理医生开处方药了。看到代码前面的缩进乱七八糟，或者看到丑陋的变量名，都会把我逼疯的。



如果黑客只是一个负责实现领导意志的技术工人，职责就是根据规格说明书写出代码，那么他其实与一个挖水沟的工人是一样的，从这头挖到那头，仅此而已。但是，如果黑客是一个创作者，他从事的就不是机械性的工作，他必须具备灵感。

黑客就像画家，工作起来是有心理周期的。有时候，你有了一个令人兴奋的新项目，你会愿意为它一天工作 16 个小时。等过了这一阵，你又会觉得百无聊赖，对所有事情都提不起兴趣。

为了做出优秀的工作，你必须把这种心理周期考虑在内。只有这样，你才能根据不同的事情找出不同的应对方法。你有一辆手动变速的汽车，你把它开上山，有时不得不松开离合器，防止汽车熄火。同样，暂时放手有时也能防止热情熄火。对于画家和黑客这样的创作者，有些工作需要投入巨大的热情，另一些工作则是不需要很操心的日常琐事。在你厌倦的时候再去做那些比较容易的工作，这是个不错的主意。

对于编程，这实际上意味着你可以把 bug 留到以后解决。消灭 bug 对我来说属于轻松的工作，只有在这个时候，编程才变得直接和机械，接近社会大众想象中的编程的样子。消灭 bug 的过程就像解一道数学题，已知许许多多的约束条件，你只要根据条件对方程求解就可以了。你的

程序应该能产生 x 结果，但是却产生了 y 结果。哪里出错了？你知道自己最后肯定能解决这个问题，所以做起来就很轻松，就好像刷墙一样，接近于休闲了。

用绘画的例子作为参考，不仅能教会我们如何管理自己的工作，还能教会我们如何与他人一起工作。历史上许多伟大的艺术品都是多人一起合作的结果，尽管最后在博物馆展出的时候，墙上可能只会写出一个人的名字。达·芬奇曾经在韦罗基奥（Verrocchio）的画室当学徒，后者当时正在画《基督的洗礼》（*Baptism of Christ*），达·芬奇的工作是完成整幅画之中的一个天使。多人一起完成一幅作品是当时的普遍做法，并不罕见。后来，米开朗基罗坚持要一个人画完罗马的西斯廷大教堂（Sistine Chapel）天顶壁画中的所有人物，他还因此被认为特别死心眼。



就我所知，当多个画家共同创作一幅作品时，每个人画的部分都是不一样的。通常来说，大师负责画主要人物，助手们负责画次要人物和背景。但是，你肯定找不到某个部分是两个人一起画的。

我认为，这也是多人共同开发一个软件的正确模式。需要合作，但是不要“合”得过头。如果一个代码块由三四个人共同开发，就没有人真正“拥有”这块代码。最终，它就会变得像一个公用杂物间，没人管理，又脏又乱，到处堆满了冗余代码。正确的合作方法是将项目分割成严格定义的模块，每一个模块由一个人明确负责。模块与模块之间的接口经过精心设计，如果可能的话，最好把文档说明写得像编程语言规范那样清晰。

就像绘画作品一样，大多数软件是为人类用户准备的。所以，黑客必须像画家一样，时刻考虑到用户的人性需要，这样才能做出伟大的产品。你必须能够站在用户的角度思考问题，也就是说你必须学会“换位思考”。

很小的时候，我就被不断告知，要设身处地为他人着想。现实中，这必然意味着你要做其他人需要的事情，而不是你自己想做的事情。这样看上去好像对我很不利，所以我暗下决心不让自己变成这样的人。

但是，我完全错了。事实表明，从他人的角度思考问题正是成功的奥秘所在。“换位思考”并不就意味着你要做自我牺牲。实际上，这是完全不同的两回事。了解别人对于事情的看法，并不代表你为他的利益服务。某些情况下，比如打仗的时候，了解对手正是为了打击对手[①]。

大多数创作者都是为人类用户而创作。为了吸引用户，你必须理解用户需要什么。举例来说，几乎所有最伟大的绘画作品都是画人的，因为人类总是对自身感兴趣的。

普通黑客与优秀黑客的所有区别之中，会不会“换位思考”可能是最重要的单个因素。有些黑客很聪明，但是完全以自我为中心，根本不会设身处地为用户考虑。这样的人很难设计出优秀软件，因为他们不从用户的角度看待问题[②]。

判断一个人是否具备“换位思考”的能力有一个好方法，那就是看他怎样向没有技术背景的人解释技术问题。我们大概都认识这样一些人，他们在其他方面非常聪明，但是把问题解释清楚的能力却惊人低下。如果聚会上，外行人问他们“什么是编程语言”，他们会这样回答：“哦，高级语言就是编译器的输入代码，用来产生目标码。”高级语言？编译器？目标码？……如果对方不知道什么是编程语言，那么他显然也不会知道这些概念。



软件的部分功能就是解释自身。为了写出优秀软件，你必须假定用户对你的软件基本上一无所知。你要明白，用户第一次使用你的软件的时候，不会预先做好功课，他们没有任何准备就开始用了，所以软件的使用方式最好能符合用户的直觉，别指望用户去读使用手册。在这方面，我见过的最佳系统是 1984 年原始的 Macintosh 电脑。它做到了那时别的软件都做不到的事情：它真的能用[③]。

---

① 下面我举一个站在对手角度考虑问题的实例。在Viaweb，每当遇到两个选项无法决策时，我们就会问自己，我们的竞争对手最恨哪一个选项？有时，对手新开发了一个基本无用的功能，但是因为他们有这个功能，而我们没有，所以他们就会在业内媒体上大肆宣传。我们当然可以解释，告诉大家这个功能是没用的，但是要是我们也开发了这个功能，就会让对手感到很恼火，所以当天下午我们就一鼓作气把自己的版本弄出来了。

② 文本编辑器和编译器不在此列。黑客开发这些软件时，不需要站在别人的立场上考虑问题，因为他自己就是典型用户。

③ 好吧，是几乎真的能用。Macintosh电脑有时会过量使用内存，导致大量对磁盘交换空间的读写（disk swapping），但是再买一个磁盘驱动器就能解决交换空间不足的问题。

源代码也应该可以自己解释自己。如果我只能让别人记住一句关于编程的名言，那么这句名言就是《计算机程序的结构与解释》[1]一书的卷首语：

程序写出来是给人看的，附带能在机器上运行。

“换位思考”不仅是为了你的用户，也是为了你的读者。这对你是有利的，因为你也会读自己写的东西。许多黑客六个月后再读自己的程序，却发现根本看不懂它是怎么运行的。我认识好几个人，因为这种经历而发誓不再使用 Perl 语言[2]。

在某些地方，自行其道、完全不替读者着想，被看成是高水平、高智商的表现，甚至都发展成了一种风尚。但是，我不觉得“换位思考”与智商之间存在任何联系。在数学和自然科学领域，你不用学习怎么向别人表达自己，也能取得很好的成就。而那些领域的人普遍很聪明，所以人们很自然地就把“聪明”与“不懂得换位思考”联系了起来。但是，世界上还有许许多多很笨的人，也同样不懂得“换位思考”。



最后，如果编程是与绘画和写作同一类的工作，黑客是否有机会像伟大艺术家一样备受推崇、流芳后世呢？毕竟生命只有一次，你可能想用它来做一些伟大的事情。

很遗憾，这个问题很难回答。声望这个东西，总是经过漫长的时滞以后才会确立，它就像遥远星系发出的光，经过了许多光年才能被我们看到。那些如今声名显赫的绘画作品，来自于五百年前的画家的卓越工作。在那些画家生前，没人像我们今天那样看重这些作品。1465 年的人们也许会感到非常奇怪，后世的人们提起乌尔比诺城（Urbino）不可一世的费德里科公爵，最主要的原因居然是弗朗切斯卡把他的鼻子画得非常独特（见图 2-2）。

---

① Harold Abelson 与 Sussman Gerald 合著的 *Structure and Interpretation of Computer Programs*，麻省理工学院出版社1985年出版。

② 把代码写得便于阅读，并不是让你塞进去很多注释。我想引申一下Abelson和Sussman的那句话：“程序写出来是为了让人看懂它的算法，附带告诉计算机如何执行。”一种好的编程语言应该比英语更容易解释软件。只有在那些不太成熟、容易出现问题的地方，你才应该加上注释，提醒读者注意那里，就好像公路上只有在急转弯处才会出现警示标志一样。



图 2-2　皮埃罗·德拉·弗朗切斯卡（Piero della Francesca）的作品《蒙特费特罗家族的费德里科》（*Federico da Montefeltro*），1465~1466 年（局部）



所以，虽然我必须承认，眼下看来艺术家比黑客更酷，但是我们不应忘记，古时候绘画蓬勃发展的那些黄金年代，画家也不是像今天这样酷的。

我们能够有把握说的就是，现在正是编程的黄金年代。大多数领域的伟大作品都诞生于很早以前。1430 年到 1500 年之间的绘画杰作，至今仍然是不可超越的。莎士比亚出现的时候，戏剧这种文艺体裁才刚发展起来，专门表演戏剧的场所“戏院”才刚刚诞生。他把戏剧这种艺术形式提升到一个难以触及的高度，让后世的每一个剧作家都不得不生活在他的阴影之中。德国雕塑大师丢勒（Albrecht Dürer）、英国小说家简·奥斯汀（Jane Austen）都是这样的大师。

我们看到这种模式一再反复出现。一种新的媒介刚刚诞生的时候，人们热情高涨、兴奋不已，短短几代人就探索清楚了这种媒介的大部分可能性，把它的能量发挥到极致。编程目前好像就处在这个阶段。

在达·芬奇的年代，绘画并不是一件很酷的事情，达·芬奇用自己的工作推动绘画成为一种伟大的表达方式。同样，编程到底能够有多酷，取决于我们能够用这种新媒介做出怎样的工作。

3

不能说的话

翻开老照片，看到以前的样子，你会不会感到难为情？我当时真的是穿成这样吗？是的，你没看错，你就是穿成这样。我们穿衣服的时候，根本不知道自己看上去有多傻，还以为很时尚。所谓“时尚”，本质上就是自己看不见自己的样子。好比我们在地球上，却感觉不到地球在动。



但真正令人惊恐的是，流行一时的不仅有衣服，还有道德观念。明明是专横武断、毫无依据的错误观点，但是大多数人却深信不疑，受到影响而不自知。这是非常危险的。流行的衣服，其实是很难看的衣服；流行的道德观念，其实不是善而是恶。但是，如果别人都穿流行的衣服，而你不穿，你就会遭到嘲讽；如果别人都遵守流行的道德观念，而你不遵守，结果则要严重得多，你会被解雇、流放、监禁，甚至被杀。

要是能坐上时间机器回到过去，不管哪一个年代，有一件事都是不会改变的，那就是“祸从口出”。你一定要小心自己说的话。自以为无害的言论会给你惹来大麻烦。今天，说地球围绕太阳运转真是再平常不过了，如果换在 17 世纪的欧洲，这么说就大难临头了。伽利略说了这样的话，结果遭到了宗教法庭的审判。①

书呆子就是那样惹上麻烦的。他们穿着不流行的衣服，讲着不合适的话。他们觉得自己说出了正确的观点，实际上却惹来了麻烦。习俗的力量不足以束缚他们。

历史的常态似乎就是，任何一个年代的人们，都会对一些荒谬的东

① 从审判一开始，宗教法庭可能就没打算对伽利略动刑，因为伽利略明确表示，不管宗教法庭提出怎样的要求，他都会照办。这是无奈之举，因为只要他不认罪，宗教法庭就不会放过他。此前不久，哲学家布鲁诺就拒不认罪，宗教法庭于是下令烧死布鲁诺。

西深信不疑。他们的信念还很坚定，只要有人稍微表示一点怀疑，就会惹来大麻烦。

我们这个时代是否有所不同？只要读过一点历史，你就知道答案几乎确定无疑，就是“没有不同”。即使有那么一丝微小的可能，有史以来第一次，我们这个时代的所有信念都是正确的，那也是出于惊人的巧合，而不是因为我们真找到了正确的方向。

一想到现在我们言之凿凿的东西，在未来人们的眼里却是荒诞不经，怎能不令人感叹呢！如果未来有人坐着时间机器来到我们这个时代，哪些话是他小心翼翼避免说出口的？我的这篇文章就想探讨这件事。我不仅要展示一些当前的“异端邪说”，让每个人都大吃一惊，还要找出适用于所有年代的通用法则，判断哪些话是不能说的。

你是一个随大流的人吗



让我先问你一个问题：大庭广众之下，你有没有什么观点不愿说出口？

如果回答是没有，那么你也许应该停下来想一想了。你的每一个观点都能毫不犹豫地说出口，你自己深深赞同这些观点，并且你也确信肯定会获得别人的赞同，这是否太过于巧合了？一种可能是，也许事情并没有这么巧合，你的观点就是从别人那里听来的，别人告诉你什么，你就相信了什么，你把别人灌输的观点当作了自己的观点。

另一种可能是，你的思想观点确实是独立思考得到的，碰巧与社会主流的思想观点一模一样。这种情况的可能性似乎不大，因为这意味着，如果别人犯错了，你也必须碰巧犯一个同样的错误。为了防止他人复制，古代制作地图的工匠会故意在地图上画错一个小地方。如果你的地图与他的地图一样，就说明不太可能是你自己独立制作的。

与历史上别的年代一样，我们的思想几乎肯定也是一张有错误的地图。如果你也犯下与别人一样的错误，那么这个错误不太可能完全来自于你自己。这就像 1972 年喇叭裤刚刚开始流行，某人声称他觉得喇叭裤很时尚，你觉得这是他完全自发产生的观点吗？

如果别人告诉你应该相信什么，你就真的相信了，那么你就会和别人一样犯下同样的错误。如果你是南北战争前的南方庄园主，你会与北

方开战；如果你是20世纪30年代的德国人，你会相信希特勒。

有时候，别人会对你说："要根据社会需要，改造自己的思想（well-adjusted）。"这种说法隐含的意思似乎是，如果你不认同社会，那么肯定是你自己的问题。你同意这种说法吗？事实上，它不仅不对，而且会让历史倒退。如果你真的相信了它，凡是不认同社会之处，你连想都不敢想，马上就放弃自己的观点，那才会真正出问题。

真　　话

到底什么话是我们不能说的？为了找到答案，首先，我们可以看看，周围的人因为说了什么而陷入麻烦。①

当然，这里要注意，并不是所有不能说出口的话都是我们要找的答案。实际上，只有同时满足两个条件才行。第一个条件是，这些话不能说出口；第二个条件是，它们是正确的，或者看起来很可能正确，值得进一步讨论。如果达不到第二个条件，大部分情况下你都不会有麻烦。你说2+2等于5，或者匹兹堡的市民身高三米，都不会有事的。这些明显错误的言论也许会被当成笑话，或者更糟一点，被当成你发疯的证据，但是肯定不会惹恼任何人。触怒他人的言论是那些可能会有人相信的言论。我猜想，最令人暴跳如雷的言论，就是被认为说出了真相的言论。



如果伽利略说帕多瓦城（Padua）的人身高三米，他只会被当作一个古怪的疯狂科学家。但是，他说地球围绕太阳运转，性质就完全不一样了。教廷知道，这种话会让人们开始思考。

回顾历史，我们会发现很多这样的例子。人们因为说真话而给自己惹来麻烦。许多的言论，今天的人们看来再平常不过，但是放在过去都是不能说的。以此推断，未来的人们很可能会发现，他们觉得很平常的话，在我们今天这个时代都是不能说的。现在有没有伽利略这样的人和

① 某些组织善意地开出一份清单，提醒你在该组织内部哪些话是不能说的。不幸的是，这种清单有两个缺点。第一个缺点是不完备，因为有些话过于惊世骇俗，开列清单者根本没想到有人真的会说出口，所以没有写入清单。第二个缺点是太笼统，清单很难真正有效实施。在一些美国大学中，校方制定了"演讲规范"（speech code），禁止人们在校园公开演讲中对种族、宗教、同性恋等问题发表歧视性的或者政治不正确的言论。如果严格按照字面含义执行这种"演讲规范"，那么就连莎士比亚也无法在美国大学中发表演讲了。

事？很可能是有的。

为了找出那些“不能说的话”，让我们问自己，它们会不会是真的？OK，当你发现某些言论很可疑时，你可以这样想，那些话听上去真是大逆不道（或者其他类似的形容词），但是有没有可能是真的？这就是找出“不能说的话”的第一种方法：判断言论的真伪。

异端邪说

除了真话以外，“不能说的话”还有一种可能。有些想法，纯粹因为非常特别，而不能说出口。比如，某个话题极其富有争议，不管是对是错，没有人敢在公开场合谈论它。我们怎样才能发现这种情况呢？

我们把这种不一定正确、但是极富争议的言论称为“异端邪说”。关注“异端邪说”，是找出“不能说的话”的第二种方法。历史上的每一个年代，都会给“异端邪说”贴一些标签，目的是在人们开始思考它们是否为真之前就把它们封杀。“亵渎神明”、“冒犯圣灵”、“异端”都是西方历史上常见的标签，当代的标签则是“有伤风化”、“不得体”、“破坏国家利益”等。以前时代的标签在今天已经不可避免地失去了杀伤力，最多只能用于讽刺。但是在以前，它们真的有巨大的威力。



举例来说，“失败主义者”（defeatist）这个词，今天看来并没有特别的政治含义，只是指某个人比较悲观，不相信自己会成功。但是在 1917 年的德国，这个词却是一件锐利的武器，鲁登道夫将军（Erich Ludendorff）将所有政治对手都称为“失败主义者”，指责他们奉行投降主义，赞成签订第一次世界大战停战协议，从而用这个借口把他们清除出政府。第二次世界大战初期，英国首相丘吉尔及其支持者也广泛使用这个词压制反对声音。1940 年那一年，任何反对丘吉尔对攻作战策略的人一律被称为“失败主义者”。这个标签贴得对不对？根本没人考虑这个问题。被贴上标签、受到打压后，人们都噤若寒蝉了。这真是很理想的压制反对声音的方法。

如今，当然也有这样的标签，从万金油式的“不适当”（inappropriate）到可怕的“制造分裂”（divisive），不胜枚举。好在不管是哪个年代，分辨这样的标签应该是比较容易的。你只要看看人们怎么称呼那些自己不赞

成、但是又不算错的观点就行了。当一个政治家说自己的对手是错的，这是直截了当的批评。但是，如果他不谈论对错，却使用“制造分裂”或者“对种族问题不敏感”这样的标签攻击对手，那么我们就应该多加注意了。

所以，如何找出那些我们自以为正确却会被未来人们耻笑的话？方法就是关注这些标签！比如，有一个标签叫做“性别歧视”，你问自己哪些想法属于“性别歧视”。然后，把头脑中跳出来的那些想法按照先后顺序列出来，再逐个追问，它们真的属于“性别歧视”吗？

这样的自问自答是不是太主观了？表面上确实很主观，但是实际上不是。因为最先从你头脑中跳出来的想法，往往就是最困扰你、很可能为真的想法。你已经注意到它们，但还没有认真思考过。

1989 年，一些聪明的研究人员设计了一个实验，给放射科医生看胸部 X 光片，请他们判断病人有没有肺癌迹象。研究人员记录了医生检查 X 光片时的眼球运动。[①]研究发现，即使那些医生漏掉了一个癌症病灶，他们的目光通常也会在那个地方停留一会。这说明他们的头脑深处已经意识到那里有问题，但是这种深层的反应没有上升为自觉的意识。我认为，类似的思维机制存在于每个人的头脑中，很多看似叛逆的“异端邪说”，早就“潜伏”在我们的思维深处。如果我们暂时关闭自我审查意识，它们就会第一个浮现出来。



时空差异

如果我们可以通晓未来，那么找出当代的那些表面上正确、实际上可笑的想法是一件很容易的事。但是，不可能做到这一点。幸运的是，我们可以找到一种几乎有同样效果的替代方法：回顾过去。我们可以去找那些过去被认为理所当然，如今却被认为不可思议的事情，这是用来找出我们自己正在犯下的错误的第三种方法。

过去和现在之间的变化有时候代表了一种进步。在物理学领域，如果我们与前人看法不一样，那是因为我们是对的，他们是错的。但是，

---

① Kundel HL、Nodine CF与Krupinski EA，“Searching for lung nodules: Visual dwell indicates locations of false-positive and false-negative decisions”，*Investigative Radiology*，24（1989），472~478。

物理学是一门硬科学[①]（hard science），换了其他学科，我们很快就无法确定谁对谁错了。如果你遇到的是社会问题，请问过去的看法与现在的看法哪一个更正确？很多时候你无法回答，因为过去与现在之间的变化往往不是因为对错，而是因为社会观念变了。比如，法定结婚年龄的变化。

我们可以自以为是地相信，当代人比古人更聪明、更高尚。但是，了解的历史越多，就越明白事实并非如此。古人与我们是一样的人，他们既不是更勇敢，也不是更野蛮，而是像我们一样通情达理的普通人。不管他们产生怎样的想法，都是正常人产生的想法。

所以，我们就有了找出“不能说的话”的第三种方法：将当代观念与不同时期的古代观念 diff [②]一下。diff 得到的结果，有一些用当代标准衡量是很令人震惊的。古人认为可以说的话，我们认为是不可以说的。但是，你有把握断言你比古人更正确吗？

甚至也不用参照历史，当代世界是多种文化并存的世界。东方文化与西方文化存在巨大的差异，各种文化的价值观念和禁忌之处都不一样。所以，你也可以将我们的观念与其他文化的观念 diff 一下。（做到这一点的最好方法，就是亲自去看一看。）

你可能会因此发现互相冲突的观点。一种文化认为，认同 x 观点是骇人听闻的，而另一种文化认为，不认同 x 观点才是骇人听闻的。不过通常来说，禁忌是单方面的。x 观点在一种文化中不允许说出口，而在其他文化中说不说都可以。我的判断是，认为 x 观点骇人听闻的一方更可能是错误的一方。[③]

有一些行为或观点，比如谋杀，在所有年代、所有地方都受到禁止

① 在学术上，“硬科学”指的是那些严格精确、以事实为依据的学科，典型代表是自然科学如物理学。相对应的概念则是“软科学”（soft science），指的是不那么严格精确、难以用事实检验的学科，典型代表是社会科学。——译者注

② diff是计算机术语，最早是一个程序，用来比较同一个软件不同版本源码之间的差异。它会告诉你，哪些是新增的代码，哪些是被删的代码。后来，这个词还可以当动词用。它最能确切表达此处我的意思。参见附录“词汇表”。

③ 有些人可能因为这一段话，认为某种程度上我是一个“道德相对主义者”（moral relativist），即不相信存在客观的、放之四海而皆准的普遍道德真理。恰恰相反，我在这里的意思是，人们自以为很“客观”，而把“主观”（judgmental）这个词用作压制讨论的标签。我们试图让自己变得“非主观”（non-judgmental）的种种努力，在未来人们看来，都将是我们的最滑稽可笑之处。



或基本禁止。我认为，只有它们才是真正错误的行为或观点。如果某个观点在大部分时空都是不受禁止的，只有我们这个社会才把它当作禁忌，那么很可能是我们出错了。

举例来说，20 世纪 90 年代早期，“政治正确”的潮流一度登峰造极。在这股潮流的推动下，哈佛大学向教职员工发了一本小册子，上面说除了其他规定以外，请尽量避免赞扬同事或学生的衣着，那样做是不合适的。“你的衬衫很不错”（nice shirt）这一类的话都不能说。我想，这种规定在全世界古往今来的各种文化之中是很罕见的。在别的地方，赞美他人的衣着更可能被看作是有礼貌的表现，而不是一种非礼。这个事件也许是一个较为温和的例子，说明了 1992 年马萨诸塞州的剑桥市[①]存在着一种古怪的禁忌。要是未来的人们坐着时间机器碰巧在那个时间来到那里，这就是他们“不能说的话”。



道貌岸然

当然，如果未来的人们真的坐着时间机器，回到马萨诸塞州的坎布里奇，他们可能需要一本特别的参考手册，里面写满了与哈佛大学有关的事情。因为那个地方的人讲究得不得了，有着许多莫名其妙的规定。那里的居民绝不容忍字母 i 上面少了一点，或者字母 t 上面少了一横。在那里，你有保证自己的每一句话都观点正确、语法无误的法定义务。这倒是提示了我们，还有第四种寻找“不能说的话”的方法：寻找那些一本正经的卫道者，看看他们到底在捍卫着什么。

孩子的大脑就是我们所有“不能说的话”的一面反射镜。我们似乎认定，孩子的思想应当是光明纯洁的。为了保证孩子不受外界“不良”思想的影响，我们对那些思想进行消毒和屏蔽，把世界描述成光明的样子，向孩子们灌输，将他们的心灵塑造成我们想象中的样子。[②]

---

① 哈佛大学所在地。——译者注

② 这种做法使得孩子对外界充满了困惑。为什么我看到的世界与大人们告诉我的世界差别如此之大？举例来说，小时候，我一直无法理解，为什么15世纪的葡萄牙“探险家”要沿着非洲海岸探险，而不是深入非洲内陆？长大后我才知道，葡萄牙人的真正目的是抓黑人充当奴隶，但是大人们不愿意孩子了解这些。参见De Azurara与Gomes Eannes合著的《几内亚的发现史》（*Chronicle of the Discovery of Guinea*），收录在Almeida编辑的《亨利王子的征服和发现》（*Conquests and Discoveries of Henry the Navigator*）一书中，George Allen & Unwin出版社1936年出版。

小孩子说脏话就是一个很好的切入点，你可以从这个小小的侧面来思考这个问题。我的许多朋友现在都开始为人父母了。他们一个个都变得非常小心，不在孩子面前使用“fuck”、“shit”这样的脏话，以免孩子学会这些词。但是，这些词是日常语言的一部分，成年人一天到晚都在用。所以，孩子从家长那里得到一个错误的印象，以为它们是没人用的。为什么家长要这样伪装呢？因为他们觉得孩子不应该知道成年人语言的所有内容，只需知道一部分适合儿童的词就行了。我们喜欢孩子们看上去天真无邪。[①]

就是因为这个原因，大多数成年人故意让孩子对世界有一个错误的认识。最鲜明的例子之一就是圣诞老人。我们觉得，小孩子相信圣诞老人，真是太可爱了。我本人其实也是这样想。但是，扪心自问，我们向孩子灌输圣诞老人的神话，到底是为了孩子，还是为了我们自己？



我在这里不讨论这样做是否正确。家长想要塑造孩子的心灵，把他们装扮成可爱的小宝宝，这可能是无法避免的。我也可能这样做。但是，就本文而言，这样做会产生一个重要结果，那就是孩子“被迫”在一个精心设计的环境中长大。他的头脑或多或少是纯洁无瑕的，一点也不知道那些“不能说的话”，从来没有被真实的社会生活“污染”过。孩子眼里的世界是不真实的，是一个被灌输进他们头脑的假想世界。将来当孩子长大以后接触社会，就会发现小时候以为真实的事情，在现实世界中是荒唐可笑的。

那些“不能说的话”就是这样被阻止进入我们头脑的。你可以想象一下，假定有一个康拉德[②]式的当代人物，他在非洲当雇佣兵，然后去了尼泊尔当医生，后来又到迈阿密经营夜总会。具体干什么并不重要，反正他就是一个见多识广的人。现在，我们把这个人的头脑，与一个在美国郊区长大的、乖巧守规矩的 16 岁女生的头脑，做一个比较。前者的所

① 很快，孩子就会从朋友那里知道这些词。但是他们明白，不能在大人面前使用。所以，没过多久，一切就变得有点像讽刺剧了。家长在外使用这些词，回家后就不用。孩子在外也使用这些词，回家后也不用。双方见面，就像演戏一样。

② 约瑟夫·康拉德（Joseph Conrad，1857—1924），著名英国小说家。17岁就开始当水手，航海生活达20余年，到过非洲和亚洲的许多地方，后来以写航海小说闻名，对英国文学产生了巨大影响。——译者注

思所想会不会令后者惊骇不已？他知道真实世界是什么样，而她知道的，或者至少体现在她言行上的，不过是父母精心灌输的一个假想世界。两者减一下，我们就可以知道不能说的到底是哪些话了。

机　　制

我还想到了第五种方法，可以找出“不能说的话”，那就是去观察禁忌是如何产生的。某种道德观念到底是怎么出现的，又是怎么被其他人接受的？如果我们能够理解它的产生机制，可能就可以应用于我们自己的时代。

流行的道德观念与其他普通的流行时尚的产生方式似乎是不一样的。一般来说，流行的时尚产生于某个有影响力的人物，他突发奇想，接着其他人纷纷模仿。15 世纪晚期，欧洲流行一种宽头鞋（broad-toed shoe），原因是当时的法国皇帝查理八世长了六根脚趾。20 世纪 20 年代，著名电影明星 Frank Cooper 决定改名，他把 Frank 改成一个印第安纳州工业小镇的名字 Gary，以便突出自己粗犷硬朗的铁汉形象，结果导致这个名字风靡一时，很多父母都为儿子取名为 Gary。但是，流行的道德观念不是这样，它们往往不是偶然产生的，而是被刻意创造出来的。如果有些观点我们不能说出口，原因很可能是某些团体不允许我们说。



那些团体神经越紧张，它们所产生的禁止力量就越大。伽利略因为宣传日心说而遭到教廷的审判，这件事讽刺的地方在于，他只是在宣传哥白尼的观点，而后者却安然无恙。事实上，哥白尼不仅不反对教廷，还是一个虔诚的天主教教士，他把自己的著作献给教皇。不幸的是，伽利略正赶上教廷内部反对派上台，宗教改革被压制，任何非正统的思想遭受到前所未有的严厉控制和禁止。

为了在全社会制造出一个禁忌，负责实施的团体必定既不是特别强大也不是特别弱小。如果一个团体强大到无比自信，它根本不会在乎别人的抨击。美国人或者英国人对外国媒体的诋毁就毫不在意。但是，如果一个团体太弱小，就会无力推行禁忌。有一种行为怪癖叫做“嗜粪症”（coprophila），它的患者人数以及影响势力眼下似乎就不太强大，无法把

自己的观点推广给其他人。

我猜想，道德禁忌的最大制造者是那些权力斗争中略占上风的一方。你会发现，这一方有实力推行禁忌，同时又软弱到需要禁忌保护自己的利益。

大多数的斗争，不管它们实际上争的是什么，都会以思想斗争的形式表现出来。16 世纪的英国宗教改革（English Reformation）本质上是为了争夺权力和财富，但是却表现为英国人要求自主的精神与罗马教廷腐化的控制之间的斗争。思想斗争更容易争取支持者。不管哪一方获胜，他们所代表的思想也就被认为获得了胜利，仿佛上帝通过选择胜利的一方表示了自己的倾向。

我并不是说斗争从来就与思想无关，而是要强调，不管实际上是否有思想斗争，斗争总是会以思想斗争的形式表现出来。正如刚刚过气的时尚并非一点儿也不时尚，失败一方的思想其实也并非一无是处。比如希特勒就很推崇写实派艺术（representational art），但是希特勒是失败者，所以写实派一直得不到认同，直到近年才开始复兴。①



虽然，流行的思想观点与流行的服饰产生方式不尽相同，但是，它们的传播途径却很相似。第一批的接受者总是带有很强的抱负心，他们有自觉的精英意识，想把自己与普通人区分开来。当流行趋势确立以后，第二批接受者就加入进来了，人数比上一批庞大得多，恐惧心在背后驱使着他们。②他们接受流行，不是因为想要与众不同，而是因为害怕与众不同。

所以，如果你要寻找“不能说的话”，可以观察流行的产生方式，试

---

① Viaweb的标志是一个红色实心圆圈，中间加上一个白色的V。我还记得，启用这个标志后没过多久，我就对实际效果感到非常满意，认为红色的圆圈是一个很有力的符号。红色可以说是最基本的颜色，圆圈是最基本的形状，它们的结合是如此具有视觉冲击力。那么，为什么很少有美国公司在标志中使用红色圆圈？嗯，我知道为什么了……（译者注：日本的国旗就是红色圆圈，而日本在二战中被美国击败。）

② 带动流行的两种力量之中，恐惧心比抱负心有力得多。好几次，我听到别人在用gyp（诈骗）这个词，我就一本正经地告诉他们，以后不能再用了，因为它是对吉普赛人（gypsy）的侮辱。但是事实上，词典写得很清楚，这两个词之间不存在词源关系，我只是在开玩笑而已。不过，别人几乎总是很诚惶诚恐地对待这个玩笑，立刻表现出一种怀着畏惧的服从。这就是流行的本质，衣着也好，思想也好，它使得人们没有自信。在新事物面前，人们会感到自己错了：这是我早就应该知道的事情啊。

着预测它会禁止哪些话。哪一个团体势力强大，却又精神高度紧张？这种团体喜欢压制什么样的思想观点？近来有没有什么社会斗争，失败的一方是哪一方，受到他们牵连的是什么样的思想观点？如果一个先锋人物想要挣脱当前的流行（比如上一代人的观点）脱颖而出，他会支持什么样的思想观点？随大流的人对什么样的思想观点抱有恐惧心？

这个方法的缺点是不全面，无法找出所有“不能说的话”。因为，我知道有些禁忌不是由于社会斗争而产生的，它们深深植根于过去的历史之中。但是，这个方法与前面四个方法结合在一起，会找出大量我们难以想象的“不能说的话”。

为什么这样做

有人可能会问，为什么要去找出“不能说的话”？为什么要故意打探那些龌龊的、见不得人的思想观点？你明知那里有挡住去路的石头，为什么还要把它们翻过来看个究竟呢？



首先，我这样做与小孩子翻石头是出于同样的原因：纯粹的好奇心。我对任何被禁止的东西都有特别强烈的好奇心。我要亲眼看一下，然后自己做决定。

其次，我这样做是因为我不喜欢犯错。如果像其他时代一样，那些我们自以为正确的事情将来会被证明是荒谬可笑的，我希望自己能够知道是哪些事情，这样可以使我不会上当。

再次，我这样做，是因为这是很好的脑力训练。想要做出优秀作品，你需要一个什么问题都能思考的大脑。尤其是那些似乎不应该思考的问题，你的大脑也要养成思考它们的习惯。

优秀作品往往来自于其他人忽视的想法，而最被忽视的想法就是那些被禁止的思想观点。举例来说，自然选择学说（natural selection）是一种伟大的理论。它的观点非常简单，你会奇怪为什么以前没有人想到。这是因为它与传统观点的差异实在太明显了，可能引发轩然大波，所以其他人不敢去想。达尔文也因此不得不非常小心，他只想当一个生物学家，不想陷入宗教争论。

在科学领域，质疑他人的结论和公认的假设是尤其重要的一件事，会

提供巨大的科学创造的优势。科学家（或者至少是优秀科学家）做事的方式，准确地说，就是寻找传统观点无法自圆其说的地方，然后试着拆开那里，看个究竟，瞧瞧里面到底出了什么问题。新的理论就是这样产生的。

换言之，一个好的科学家，并不仅仅是避开传统观点，还要努力打破传统观点。科学家就是要自找麻烦。这应该是任何学者的研究方式，但是科学家似乎特别愿意一探究竟。

为什么？可能仅仅是因为科学家比其他领域的学者更聪明。如果有必要的话，大多数物理学家有能力拿到法国文学的博士学位，但是反过来就不行，很少存在法国文学的教授有能力拿到物理学的博士学位。[①]或者，另一种原因是，在科学中，命题的真伪更显而易见，所以这使得科学家能够更勇敢地质疑传统观点。（这句话也可以这样说，因为科学命题的真伪更显而易见，所以你想在科学界谋职，就不得不训练自己的智力，去发现解决那些真正的问题，而不能仅仅当一个政治家，通过搞人事关系和派系斗争立足。）



不管是哪一个原因，看来存在一个很清晰的关联关系：智力越高的人，越愿意去思考那些惊世骇俗的思想观点。这不仅仅因为聪明人本身很积极地寻找传统观念的漏洞，还因为传统观念对他们的束缚力很小，很容易摆脱。从他们的衣着上你就可以看出这一点：不受传统观念束缚的人，往往也不会穿流行的衣服。

做一个异端是有回报的，不仅是在科学领域，在任何有竞争的地方，只要你能看到别人看不到或不敢看的东西，你就有很大的优势。眼下的

---

① 这句话本身就是一种明显的本文所讨论的“不能说的话”。它犯了大学中的一个大忌：评判各种学科的难易。大学校园中有一条默认的公理——各种领域的研究所要求的智力水平都是相同的。毫无疑问，这条公理确实能够减少冲突，让一切平稳运作。但是，如果这条公理为真，那将是多么巧合的事情啊，所有学科的难易程度居然一模一样！而且，承认这条公理比不承认它会使得一切都方便得多！你只要想到这些，怎能不质疑它呢！尤其是当你想到，一旦接受了这条公理所产生的必然推论，就更无法不质疑它了。比如，它意味着不会出现单个学科的停滞或爆发式发展，所有学科的发展形态必须是完全同步的，因为这条公理告诉我们，各个学科面对的问题难度是一样的！（要弥补这个推论，你真的会伤透脑筋。）

此外，如果大学开设了烹饪系或运动管理系（sports management），你会怎么想？如果你接受上面的公理，那么大学到底还要开设什么系？你真的认为微分几何和烹饪学的难度相同吗？

美国汽车工业对于市场份额下降怨天尤人。但是，这件事再明显不过了，任何人只要略做观察，就能迅速说出美国汽车公司走下坡路的原因：它们生产烂车。更糟的是，长期以来，它们一直这样做，所以现在美国车完全是在吃品牌的老本，也就是说，消费者购买凯迪拉克汽车，不是因为汽车本身，而是因为它的品牌。实际上，现在的凯迪拉克早已不是1970年时的凯迪拉克了。但是，我想没人敢这么说。[①]否则，这些汽车公司早就把问题解决了。

训练自己去想那些不能想的事情，你获得的好处会超过所得到的想法本身。这就像田径比赛之前要做一些伸展运动，把肢体活动开。你要把身体伸展到极限，远超过跑步所需要的那种程度，这样一来，比赛的时候才能跑得更快。同样，如果你能“远远地”跳出传统思维，提出让别人一听就脑袋轰一声炸开的惊人观点，那么你就在“小小地”跳出传统思维方面不会有任何困难。要知道，人们把后面的这种情况称为“创新”。



守口如瓶

一旦发现了“不能说的话”，下一步怎么办？我的建议就是别说，至少也要挑选合适的场合再说，只打那些值得打的仗。

假设未来的某一天，世界上爆发了一场运动，黄颜色被禁止了。任何东西都不得涂成黄色，违者就是“黄色分子”（yellowist），以破坏社会稳定罪论处。橙色可以容忍，但也很可疑。有一天，你终于觉醒了，意识到错的不是黄颜色，而是这个社会。如果公开这样说，就会被打成“黄色分子”，无数正义人士义愤填膺，对你口诛笔伐。如果你以此作为人生目的，一定要为黄颜色平反昭雪，现在的局面可能正中你下怀。但是，如果你的兴趣主要是别的事情，变成他人眼里的“黄色分子”对你

① 在这些公司内部，类似的想法很可能被贴上“悲观消极”、“失败分子”这样的标签。优秀的决策者根本不应该在乎这些标签，而是直接问自己，它们到底对不对？其实，一个公司是否健康运作，可以用一个指标衡量，那就是对负面评价的容忍程度。做出伟大产品的公司，自我评价往往以“批评”和“自嘲”为主，而不是以“肯定”和“表扬”为主。我认识的杰出成就人士都认为自己做得不好，之所以能成功只是因为其他人做得更差。

是极大的干扰。与笨蛋辩论，你也会变成笨蛋。

这时你要明白，自由思考比畅所欲言更重要。如果你感到一定要跟那些人辩个明白，绝不咽下这口气，一定要把话说清楚，结果很可能是从此你再也无法自由理性地思考了。我认为这样做不可取，更好的方法是在思想和言论之间划一条明确的界线。在心里无所不想，但是不一定要说出来。我就鼓励自己在心里默默思考那些最无法无天的想法。你的思想是一个地下组织，绝不要把那里发生的事情一股脑说给外人听。“搏击俱乐部”的第一条规则，就是不要提到搏击俱乐部。[①]

1638 年，英国诗人弥尔顿（John Milton）准备第一次访问意大利。曾经担任英国驻威尼斯大使的沃顿爵士（Henry Wootton）告诉弥尔顿要记住一句座右铭“i pensieri stretti & il viso sciolto”。字面意思是“守口如瓶，笑脸相迎”，也就是说，你要对每一个人微笑，但是不要说出自己的真实想法。这是很明智的建议。因为弥尔顿是一个喜欢争论、好打嘴仗的人，而当时罗马教廷的宗教裁判所非常强势，所以沃顿爵士才会这样建议他。需要记住的是，弥尔顿的时代与我们的时代并没有本质不同。每个时代都有自己的忌讳，如果你触犯它们，就算没有坐牢，至少也会为自己惹来麻烦，干扰了正常生活。



我承认，“守口如瓶”看上去是一种怯懦的行为。每当我读到山达基教会（Scientology）的信徒对批评者骚扰不断，[②]或者抗议以色列侵犯人权的人士被贴上“反犹太人”的标签，[③]或者研究人员受到 DMCA[④]诉讼威胁，[⑤]我内心就有一个声音在高喊：“好吧，你们这些混蛋，让我们来说清楚。”可是问题在于，“不能说的话”太多了，如果口无遮拦，你就没时间做正事了。为了与他人论战，你不得不变成一个语言学家，比如

① 《搏击俱乐部》（*Fight Club*）是1999年的美国电影，讲述了一个地下组织发起人的故事。在电影中，加入“搏击俱乐部”的第一条规则就是不得谈论搏击俱乐部。——译者注

② Richard Behar，“The Thriving Cult of Greed and Power”，*Time*，1991年5月6日。

③ Patrick Healy，“Summers hits ‘anti-Semitic’ actions”，*Boston Globe*，2002年9月20日。

④ DMCA指美国的《数字千禧年版权法》（Digital Millennium Copyright Act），该法律主要保护版权作品的互联网传播权。DMCA规定，如果某个网站侵犯了你的版权，你可以向网站所有者或者主机服务商发出通知，要求撤下侵权内容，这个通知就叫做“DMCA通知”。如果对方没有及时采取行动，你就可以把它告上法庭。——译者注

⑤ “Tinkerers’ champion”，*The Economist*，2002年6月20日。

诺姆·乔姆斯基。[①]

“守口如瓶”的真正缺点在于，你从此无法享受讨论带来的好处了。讨论一个观点会产生更多的观点，不讨论就什么观点也没有。所以，如果可能的话，你最好找一些信得过的知己，只与他们畅所欲言、无所不谈。这样不仅可以获得新观点，还可以用来选择朋友。能够一起谈论“异端邪说”并且不会因此气急败坏的人，就是你最应该认识的朋友。

笑脸相迎?

你的策略，简单说，就是不赞同这个时代的任何一种歇斯底里，但是又不明确告诉别人到底不赞同哪一种歇斯底里。狂热分子试图引诱你说出真心话，但是你可以不回答。如果他们不放手，一定要你回答“到底是赞成还是反对我们”，你不妨以不变应万变：“我既不反对也不赞成。”



不过，更好的回答是“我还没想好”。哈佛大学校长拉里·撒墨尔斯（Larry Summers）被逼表态时，就是这样说的。[②]他后来解释说：“别想在我身上做石蕊试验[③]。”人们喜欢讨论的许多问题实际上都是很复杂的，马上说出你的想法对你并没有什么好处。

假设社会上充斥着反对“黄色分子”的人，他们只要看谁不顺眼，就大肆攻击。你看不下去，准备出手反击。这时，有几种方法可以使你免于被贴上“黄色分子”的标签。你可以参考战争史上的局部战争案例，避免正面对抗敌人的大部队，只打一些小规模的局部战争。比如，从远处用弓箭骚扰他们就是很好的方法。

具体来说，一种方法就是逐步把辩论提升到一个抽象的层次。假定总的来说，你反对言论审查制度。公开质疑的时候，你一定要小心，不要提到具体的被审查的电影或者书籍。否则，对手就会一把抓住那部电

① 我这里不是指你一定要持有诺姆·乔姆斯基的观点，而是指你不得不变成一个专业的辩论者（controversialist）。如果说了“不能说的话”，你就同时得罪了保守派和自由派，两派都会与你辩论。好比回到维多利亚女王时代的英国，你同时得罪了辉格党和托利党，那么你的一张嘴怎么能是两大党派的对手呢？（编者注：诺姆·乔姆斯基是著名的语言学家，麻省理工学院的语言学和哲学教授。）

② James Traub，“Harvard Radical”，*New York Times Magazine*，2003年8月24日。

③ 石蕊是一种化学液体，遇到酸性物质时变红，遇到碱性物质时变蓝。所以，石蕊试验通常用来判断某种物质的酸碱性。——译者注

影或那本书籍，声称你支持的其实不是言论自由，而是那些被审查的内容。你不要直接攻击某个标签，而要攻击它的“元标签”（meta-label）。所谓“元标签”，就是对某个标签的抽象描述。如果人们开始讨论元标签，那么原来的标签反而不会受到注意了。举例来说，“政治正确”（political correctness）就是一个“元标签”，是许多特定现象的总称。这个词现在被广泛使用，其实这恰恰意味着“政治正确”的时代正在开始消亡，因为它使得你可以从总体上攻击这个现象，而不会受到指控，不会被说成支持某一种特定的“政治不正确”现象。

另一种反击的方法就是使用隐喻（metaphor）。20 世纪 50 年代，美国众议院的“非美委员会”（Un-American Activities Committee）以遏制共产主义为名，大肆迫害文艺界和政治界的进步人士。剧作家阿瑟·米勒创作了戏剧《萨勒姆的女巫》①（*The Crucible*）进行反击。虽然在戏中他一句也没有提到“非美委员会”，但是观众一眼就可以看出，他在讽刺现实，将搜捕共产党间谍比喻为莫须有的捉女巫。“非美委员会”根本无法做出回应，你总不能为审判女巫辩护吧？阿瑟·米勒的隐喻太贴切了，直到今天，“非美委员会”的行为还经常被描述为“搜捕女巫”（witch-hunt）。

所有反击方法之中，最好的一种可能就是幽默。狂热分子都有一个共同点：缺乏幽默感。他们无法平静地对待笑话。在幽默王国中，他们闷闷不乐，就像满身笨重盔甲的骑士走进了溜冰场，无所适从。一个现实的例子就是，维多利亚女王时代的英国人讲究宫廷礼仪，迂腐守旧，人们把这当作笑话看待，结果它好像就真的被笑话击垮了。它在当代的化身“政治正确”也将得到同样的命运。“我很高兴自己写了《萨勒姆的女巫》，”阿瑟·米勒写道，“但是回想起来，我常常希望自己有那种气质，写一出反映当时情况的荒诞喜剧。”②



---

①《萨勒姆的女巫》是阿瑟·米勒1953年的作品，讲述了1692年的北美马萨诸塞州萨勒姆小镇，出现了对于女巫的恐慌，从而进行了一场荒诞残酷的审判。许多善良的人由于他人的陷害和莫须有的罪名被诬陷入狱，在法庭上无法承受巨大的压力和威胁，不得不违心地承认自己有罪。阿瑟·米勒通过这部作品，影射当时美国国内“麦卡锡主义者”对进步人士的迫害，就像历史上对女巫的审判一样荒唐。——译者注

② Arthur Miller，*The Crucible in History and Other Essays*，Methuen，2000年。

永远质疑

一个荷兰朋友建议我，把荷兰作为具有宽容精神的社会的例子。没错，历史上，荷兰人确实长期具有相对开放的思想。几个世纪以来，这个地势低洼的欧洲国家一直是言论相对自由的地方。在那里，你可以放心说出其他地方不能说的话。这帮助它成为学术和工业的中心。（言论自由与这两者紧密结合的历史，比大多数人意识到的还要长。）哲学家笛卡儿虽然被认为是法国人，但是他的思想大部分是在荷兰境内形成的。

但是，我还是怀疑。荷兰这个国家到处都是法规和管制，有许许多多的事情都是明确禁止的。在这种情况下，你真的可以畅所欲言吗？

荷兰人认为自己思想开放，但是这种想法本身却什么也证明不了。有谁认为自己的思想不开放？美国郊区的中产阶级白人家庭普遍家教严格，限制孩子与外界多接触，可是在那里长大的女孩子也认为自己思想开放呢。不管问谁，人们都会说同样的话："我们心态很开放，愿意接受新思想。"但是实际上，人们脑子里有一根界线，早就认准了什么是对的，什么是错的。[①]换言之，在他们看来，所有观点都是可以讨论的，除了那些错的观点。



如果你的数学不好，那么你自己会知道，因为考试的时候你得不出正确答案。但是，如果你的思想很保守，你自己不会知道，而且你很可能还会持有相反的看法。请记住，所谓"流行"（传统观念也是一种流行），本质上就是自己看不见自己的样子。否则就不会有流行了。对于那些被流行抓住的人，流行就不再是流行，而是应该要做的正确事情。只有保持一定的距离才能观察到人们观念的变化，发现流行（也就是人们自以为正确的事情）到底是什么。

时间就是一种产生距离的简单方法。实际上，新的流行让旧的流行更容易被观察到，因为对比之下，旧的流行会显得很荒唐。从钟摆波动的一端望去，上一个周期的端点就显得特别遥远。

不过，想要摆脱你自己的时代的流行，需要一点自觉。没有了时间所产生的距离，你不得不自己创造距离。你不要让自己成为人群的一分子，

① 有些地方的人表面上不说你的观点是错的，而是使用更婉转的中性词来表达自己的判断，比如"负面的（观点）"或者"有破坏作用的（观点）"。

而要尽可能地远离人群，观察正在发生的事情，特别注意那些被压制的思想观点。比如，有些软件提供“互联网过滤”功能（Web filter），防止孩子和雇员看到色情的、暴力的、宣扬仇恨的网站。什么才算是色情和暴力？什么叫做“宣扬仇恨”？这种功能听上去很像出自小说《1984》[①]。

各种各样的标签可能是外部线索的最大来源，帮助你发现这个时代流行的是什么。如果一个命题是错的，这就是它所能得到的最坏评价，足够批判它了，根本不用再加上任何其他标签。但是，如果一个命题不是错的，却被加上各种标签，进行压制和批判，那就有问题。因为只要不是错的观点，就不应该被压制讨论。所以每当你看到有些话被攻击为出自××分子或××主义，这就是一个明确的信号，表明背后有问题。不管在 1630 年还是在 2030 年，都是如此。当你听到有人在用这样的标签，就要问为什么。

如果你发现自己也在用这些标签，那就更要问为什么。你不仅要远距离观察人群，更要远距离观察你自己。顺便提一句，这可不是激进的想法，儿童和成年人的主要差别就在这里。儿童精疲力竭时，可能会大发脾气，因为他不知道为了什么；成年人则会了解是个人的身体状况问题，与外界无关，说一句“没关系，我只是累了”。我想，通过类似的机制，一个人完全可以识别和抵制外界流行的道德观念，把它们与内心世界相分离。



如果你想要清晰地思考，就必须远离人群。但是走得越远，你的处境就会越困难，受到的阻力也会越大，因为你没有迎合社会习俗，而是一步步地与它背道而驰。小时候，每个人都会鼓励你不断成长，变成一个心智成熟、不再耍小孩子脾气的人。但是，很少有人鼓励你继续成长，变成一个怀疑和抵制社会错误潮流的人。

如果自己就是潮水的一部分，怎么能看见潮流的方向呢？你只能永远保持质疑。问自己，什么话是我不能说的？为什么？

---

①《1984》是英国左翼作家乔治·奥威尔（George Orwell，1903—1950）的代表作。在这篇小说中，作者以辛辣的笔触批判了极权主义，讽刺了泯灭人性的追逐权力的人。

——编者注



4 良好的坏习惯

在大众眼里，“黑客”（hacker）就是入侵计算机的人。可是，在程序员眼里，“黑客”指的是优秀程序员。这两个含义其实是相关的。对于程序员来说，“黑客”这个词的字面意思主要就是“精通”，也就是他可以随心所欲地支配计算机。



更麻烦的是，“黑”（hack）这个词也有两个意思，既可以用作赞美，也可以用作羞辱。如果你解决问题的方式非常丑陋笨拙，这叫做你很“黑”。如果你解决问题的方式非常聪明高超，将整个系统操纵在股掌之间，这也叫做你很“黑”。[①]日常生活中，前一种意思更多见，可能因为丑陋的做法总是多于聪明的做法。

信不信由你，“黑”的这两个意思也是相关的。丑陋的做法与聪明的做法存在一个共同点，那就是都不符合常规。你用胶带把包裹绑在自行车上，那是不符合常规的丑陋做法；你提出充满想象力的新概念，推翻欧几里得空间（Euclidean space），那是不符合常规的聪明做法。从“丑陋”到“聪明”，它们之间存在一种连续性渐变。

早在计算机出现之前，黑客就存在了。费曼[②]为曼哈顿计划工作时，喜欢破解存放机密文件的保险箱，觉得这样很有趣。这种传统持续至今。读研究生时，我有一个黑客朋友，他费尽心力配齐了一整套的开锁工

① 中文的“黑”很难体现这两个意思，而在英文中，hack prose意思是平庸陈腐的文章，而hack the problem意思是很漂亮地解决了一道难题。——译者注

② 费曼（Richard Feynman，1918—1988），美国著名物理学家，诺贝尔奖得主，以性格顽皮、特立独行著称。——译者注

具。[1]（现在，他在管理一个对冲基金，那个行业与开锁并非毫无关系。）

有时，你很难向当局解释为什么有人喜欢做这种事。我的另一个朋友，曾经因为入侵计算机，受到了政府的调查。最近，这种行为已经被认定为一种犯罪，但是联邦调查局发现，通行的调查方法不适用于黑客。警方总是从犯罪动机开始调查。常见的犯罪动机不外乎毒品、金钱、性、仇恨等。满足智力上的好奇心并不在FBI的犯罪动机清单之上。说实话，这个概念对他们来说完全陌生。

总体来看，黑客是不服从管教的，这往往会激怒管理当局。但是，不服从管教，其实是黑客之所以成为优秀程序员的原因之一。当公司的CEO装模作样发表演说时，他们可能会嘲笑他；当某人声称某个问题无解时，他们可能也会嘲笑他。如果硬要他们服从管教，他们也就无法成为优秀程序员了。

不过，有些人的这种态度不是真的，而是装出来的。某些年轻程序员注意到了知名黑客的怪癖，就会模仿，好使自己显得更聪明。这种装出来的不服从再加上故作姿态挑毛病的态度，不仅仅令人恼火，而且实际上会延缓创新的进程。

但是，即使考虑到黑客令人恼火的种种怪癖，他们不服从管教的性格依然是利大于弊。我希望人们能理解，能更多地看到这种性格的长处。

举例来说，好莱坞的电影人一直大惑不解，为什么黑客不喜欢版权



---

① 我也曾经打算学习开锁，但是原因不仅仅是为了满足好奇心，或者磨练自己的智力。那时，我读研究生读到一半，每天都要上机。管理机房的本来是一些本科生，他们很聪明也很不安分，后来就被换掉了，改成一个专职的机房管理人员。他每天下午5点锁好机房，准时下班回家。如果在此之后计算机出问题了，理论上你就只能等到第二天早上他来上班时再重启机器。这种做法根本不可行，因为我们这些研究生往往到下午5点才会开始上机干活。幸运的是，哈佛大学计算机系的Aiken实验室（现在已废弃）楼层之间有隔层，通向一扇天花板上的暗门正对着机房管理员办公室。我们就直接从这扇暗门进入屋里，打开管理员的抽屉，拿到机房钥匙。

有一天晚上，大概凌晨3点，我从天花板爬到管理员的桌子上，震耳欲聋的警报声突然响彻整幢大楼。“Fuck，”我心想（抱歉用了这个不雅的词，但是我清楚地记得，自己当时就是这么想的），“他们装了警报器。”我在三十秒内仓皇逃出大楼，冒着倾盆大雨，一路跑回家。虽然我装出一付若无其事的样子，但是内心却惊恐地觉得自己是一个犯罪分子，身边的每一辆汽车仿佛都是警车。第二天，我把辩解的理由排练得滚瓜烂熟后，才回到实验室。但是，出乎意料，并没有追究责任的Email在等着我。事实上，昨晚暴风雨，闪电大作，触发了警报器。

法。在黑客网站 Slashdot 上面，版权是永恒的讨论热点。为什么程序员那么关心版权，而不是其他事情?

部分原因是，有些公司为了防盗版而使用了禁止复制的技术。这等于交给黑客一把锁，他的第一反应肯定是如何才能打开它。但是，这里面还有更深层次的原因，对于版权和专利这样的制度，黑客深感担忧。他们感到，保护“知识产权”的力度不断增大，已经威胁到了他们完成工作所必需的“思想自由”[①]。在这一点上面，他们的看法是正确的。

只有深入了解当前的技术，黑客才能构想下一代技术。知识产权的拥有者也许会说，不，谢谢，我们不需要你的帮助，我们自己就能开发下一代技术。他们错了，在计算机工业的历史上，新技术往往是由外部人员开发的，而且所占的比例可能要高于内部人员。1977 年，IBM 公司内部肯定有一些部门正在开发下一代电脑。他们没有料到的是，真正的下一代电脑不是诞生于 IBM 实验室，而是由两个与他们完全不相干的长头发年轻人在旧金山的一间车库里开发出来的。这两个年轻人，一个是史蒂夫·乔布斯，另一个是史蒂夫·沃兹尼亚克[②]（图 4-1）。差不多同一时间，计算机工业的几大巨头聚在一起，合作研发官方版的下一代操作系统 Multics。但是，另外两个年轻人——26 岁的肯·汤普森和 28 岁的丹尼斯·里奇——觉得 Multics 过分复杂，就另起炉灶，写出了一个自己的操作系统。他们参照 Multics，为它取了一个搞笑式的名字 Unix[③]。



最新的版权法设置了前所未有的障碍，禁止外部人员了解专有技术的内部细节，从而也就禁止了外部人员从这个途径产生新构想。过去，厂商使用专利，防止你出售他们产品的复制品，但是他们无法阻止你把产品拆开，了解内部的工作原理。最新的版权法将后面的这种行为定义为一种犯罪。如果我们不可以研究当前的技术，不能思考如何改进它，那么我们怎样才能开发出新技术呢?

---

① 思想自由（intellectual freedom），指的是自由思考以及表达这种思考的权利。它是《世界人权宣言》（*Universal Declaration of Human Rights*）第19条规定的一种人权。参见 http://en.wikipedia.org/wiki/Intellectual_freedom。——译者注

② 苹果电脑公司的两个创始人。1977年，苹果公司推出的APPLE II计算机是世界上第一台个人电脑。——译者注

③ 在英语中，前缀Multi-意思是“多个”，而前缀Uni-意思是“单个”。——译者注



图 4-1　1975 年，乔布斯和沃兹尼亚克

具有讽刺意味的是，这种局面正是黑客自己造成的。计算机是《版权法》修改的根本原因。历史上，机器内部的控制系统一直是物理装置：齿轮、杠杆和连接器等。但是，计算机的出现使得机器的控制系统逐渐变成了软件，产品的价值也由软件来决定。[①]我这里所说的软件是一个统称，包括数据（data）在内。胶木唱片上的歌曲属于数据，是用物理方法压制在塑料盘片上的；iPod 里的歌曲也属于数据，它储存在硬盘里面。

数据在本质上就是容易复制的。互联网的出现使得复制品更容易流通。难怪那些公司感到害怕了。但是，如同往常一样，恐惧影响了他们的判断。他们推动政府通过了严厉的法律，保护知识产权，作为对新技术的回应。立法者的原意可能是好的。但是，他们也许没有意识到，这样的法律弊大于利。

为什么程序员如此激烈地反对这样的法律？如果我是立法者，肯定对这种神秘现象有兴趣。这就好比如果我是一个农夫，半夜突然听到鸡舍有动静，肯定会去看个究竟。黑客都是聪明人，很少出现所有人意见一致的情况。如果他们都说有问题，那么也许真的就是什么地方出了问题。

那些法律有没有可能是错的？虽然它们原意是维护美国的利益，但是实际上却适得其反？想一想吧，理查德·费曼喜欢破解保险箱，真的很符合美国人性格。很难想象，当时的德国政府会有同样的幽默感。也

① 日益软件化的，并不仅仅是产品的内在。由于制造业自动化程度越来越高，产品的外在也逐渐由软件决定了。



许这不是巧合。

黑客是不服从管教的，这就是他们的本性。这也是美国人的本性。硅谷出现在美国，而不是出现在法国、德国、英国、日本，这绝非偶然。后面那些国家的人们总是按部就班地行事。

我曾经住在意大利的佛罗伦萨。住了几个月以后，我发现自己内心真正寻找的地方其实是我刚刚离开的地方。佛罗伦萨之所以著名，完全是因为这个城市在 1450 年的显赫地位，它是那时的纽约，形形色色疯狂而有抱负的人们来到这里。现在，这样的人都去了美国。（所以，我又回到了美国。）

对于适当的不服从管教，保持宽容不会有太大的坏处，反而很有利于美国的国家优势，它使得美国不仅能吸引聪明人，还能吸引那些很自负的人。黑客永远是自负的。如果黑客有自己的节日，那就是 4 月 1 日愚人节，你可以放心地作弄其他人。黑客的这种自行其是的特点，很大程度上说明了，为什么不管是出色的工作还是糟透了的工作，黑客都用同一个词形容[①]。如果他们做出了一个东西，他们自己总是无法百分百确定那到底是什么东西。有可能完全没用，但是只要那些出错的地方还算正常，那么就是一个信号，表明这个东西还有希望。在人们心目中，编程是非常精确、有条不紊的，这真是非常奇怪的想法。计算机确实是非常精确、有条不紊的，但是黑客的所作所为完全出于兴趣，想到哪里就做到哪里，没有明确的计划，只求开心。



在黑客世界中，有些最典型的解决问题的方法实际上与玩笑也相差不远。IBM 推出个人电脑的时候，懒得自己开发操作系统，就与微软公司签了一个很大方的授权协议，将微软的 DOS 作为默认操作系统，每卖出一台电脑，微软都可以提成，并且还可以把 DOS 授权给其他公司使用。这份授权协议的结果无疑让 IBM 感到非常吃惊。另一个例子是 Michael Rabin[②]遇到难题的时候，会把问题重新定义成一个较简单的形式，同时一定会假想一个对手正在与他比赛谁能更快地解决问题。

---

① 此处就是指文章开头提到的“黑”（hack）这个词。——译者注

② Michael Rabin（1931—），以色列计算机科学家，1976年的图灵奖得主。——译者注

很自负的人必须培养出敏锐的感觉，及时发现周围情势的变化，知道怎样才能脱身。最近，黑客就感觉不太对，大气候变了。对于不服从管教，政治气氛变得严厉了。[①]

近来一系列的政策变化，使得这个国家的公民自由范围不断收缩减小。对于黑客来说，这是非常不好的兆头。普通人肯定会感到大惑不解，为什么黑客如此在乎公民自由？为什么程序员会比牙医、销售员、园艺师更在乎呢？

让我以政府官员听得懂的语言来解释这件事情。公民自由并不仅仅是社会制度的装饰品，或者一种很古老的传统。公民自由使得国家富强。如果将人均国民生产总值与公民自由的关系画成图，你会发现它们是很清楚的正相关关系。公民自由真的是国家富强的原因，而不是结果吗？我认为是的。在我看来，一个人们拥有言论自由和行动自由的社会，往往最有可能采纳最优方案，而不是采纳最有权势的人提出的方案。专制国家会变成腐败国家，腐败国家会变成贫穷国家，贫穷国家会变成弱小国家。经济学里有一条拉弗曲线（Laffer curve），认为随着税率的上升，税收收入会先增加后减少。我认为政府的力量也是如此，随着对公民自由的限制不断上升，政府的力量会先增加后减小。[②]至少现在看来，我们的政府很可能蠢到会真的把这个实验付诸实施，亲自验证一下这个观点。但是，税率提高了还能再降下来，而一旦这个实验铸成大错，就悔之晚矣，因为极权主义制度只要形成了，就很难废除。



这就是为什么黑客感到担忧。政府侵犯公民自由，表面上看，并不会让程序员的代码质量下降。它只是逐渐地导致一个错误观点占上风的世界。黑客对于公民自由是非常敏感的，因为这对他们至关重要。他们远远地就能感到极权主义的威胁，好比动物能够感知即将来临的暴风雨。

如果正如黑客所担忧的，近来那些旨在保护国家安全和知识产权的措施最终却成为一枚导弹，不偏不倚瞄准了美国的优势所在，那可真是

---

① 2001年9・11事件以后，美国通过了“爱国者法案”，以防止恐怖主义为目的大大扩张了警察机关的权限。那些不遵守法规的可疑分子将受到比以前严厉得多的审查和惩罚。
——译者注

② 我很乐意将我的名字用来命名这条曲线，命名后更容易记住这个观点。

太讽刺了。不过，这种事情早有先例：人们惊慌失措时采取的措施到头来产生了适得其反的效果。

有一种东西，叫做美国精神（American-ness），生活在国外的人最能体会到这一点。如果你想知道哪些事情可以滋养或者削弱这种精神，不妨去问问黑客，他们是最敏感的焦点人群，因为在他们身上，比我知道的其他人群，更能体现出这种精神。真的，他们可能比那些政府里掌管美国的人更懂得什么叫做美国精神。那些政客开口必谈爱国主义，总是让我想起黎塞留[①]（Richelieu）或者马萨林[②]（Mazarin），而不是杰弗逊或者华盛顿。

如果读美国开国元勋的自述，你会发现他们听起来很像黑客。“反抗政府的精神，”杰弗逊写道，“在某些场合是如此珍贵，我希望它永远保持活跃。”



你能想象今天的美国总统也这么说吗？这些开国元勋就像直率的老祖母，用自己的言辞让他们的那些不自信的继承者感到了惭愧。他们提醒我们不要忘记自己从何而来，提醒我们，正是那些不服从管教的人们，才是美国财富与力量的源泉。

那些占据高位、本能地想要约束黑客、强迫黑客服从的人们，请小心你们的要求，因为你们真有可能成为千古罪人。

---

① 黎塞留（1585—1642），法国政治家，路易十三的主要大臣，以具有政治谋略、玩弄政治手段著称，对当时法国集权制度的形成和巩固发挥了重要作用。——译者注

② 马萨林（1602—1661），法国政治家，继承黎塞留，担任路易十四的主要大臣，也是一个政治斗争能力强于治国能力的政治家。——译者注

5

另一条路

1995 年的夏天，我和我的朋友罗伯特·莫里斯[①]决定创业。那时，正赶上网景公司[②]的股票即将上市，各种公关活动如火如荼，媒体都在谈论网络商务。当时大约有 30 家左右的网上商店，全部是手工制作网页。如果网络商务真要大规模发展，必须用专门的软件制作网上商店，所以我们决定动手写这样的软件。

第一周，我们打算写一个传统的桌面软件。没过几天，我们就想了另外一个方法：可以让软件在服务器上运行，浏览器作为操作界面。我们试着重写，让所有操作都通过网络完成。那时已经很清楚了，这就是我们的方向。如果软件运行在服务器上，一切都会简单得多，无论对于用户，还是对于我们。

事实证明我们是正确的。我们的公司后来被雅虎收购，现在的名字是 Yahoo! Store。我们的软件是最受欢迎的网上商店生成器，用户超过 2 万人。

创立 Viaweb 的时候，我们对别人说软件运行在服务器上，几乎没人懂这是什么意思。直到一年后，Hotmail 开始运作[③]，人们才有点儿明白这个概念。现在，大家都知道了这是一个可行的方法，而且我们还有了

① Robert Morris（1965—），现任麻省理工学院计算机系副教授。1988年，写出了网上第一个蠕虫程序；2005年，与作者共同创立了风险投资公司Y Combinator。——译者注

② 网景公司的股票于1995年8月9日上市，发行价格28美元一股，盘中最高价格为74.75美元，报收在58.25美元，涨幅高达208%。这直接促成了网络创业热，带动了雅虎、eBay、亚马逊的上市，开创了互联网公司的上市热潮。——译者注

③ Hotmail于1996年7月4日上线，是第一批浏览器界面的电子邮件服务提供商。它首创了用户可以在邮件中使用HTML标记语言（因此Hotmail的原始写法是HoTMaiL），并且每位用户有2MB的免费存储空间。——译者注



专门的名字：应用服务供应商（Application Service Provider），简称 ASP。

我认为，大量的下一代软件都将采用这个模式。甚至最大的输家——微软公司，看来也明白了，部分软件从桌面消失将是不可避免的。如果软件从桌面移到服务器上，对于开发者来说，一切将发生根本性的变化。本文将站在先行者的角度，描述我们正在经历的种种令人惊叹的变化。由于软件转移到服务器的趋势才刚刚开始，所以我下面所写的是对未来的憧憬。

下一个潮流？

回顾桌面软件时代，我想我们会为人们忍受的种种不便惊叹不已。这就好像汽车刚诞生的时候，车主忍受的不便会令现在的人们惊叹不已一样。汽车诞生的头二三十年，你想当车主，非得成为汽车专家不可。但是汽车用处太大了，很多不是专家的人也想拥有它们。



计算机现在就处在这个阶段。一旦拥有了桌面电脑，你就被迫不情愿地学习很多东西，了解它的内部运作机制。超过一半的美国家庭拥有电脑，我妈妈就有一台，用来收发邮件和记账。几年前，她收到苹果公司的一封来信，上面说她可以优惠购买新版的操作系统。老太太被这些术语吓坏了。一个 65 岁的妇女，只用电脑收发邮件和记账，却被迫要和操作系统打交道，搞清楚要不要安装一个新版本，这真是太过分了。普通用户根本没必要知道“操作系统”这个词，更不要说“驱动程序”和“补丁”了。

现在，可以有另一种方法发布软件，用户再也不会被迫当上系统管理员了。互联网软件运行在服务器上，用户界面就是网页。对于普通用户来说，这种新型软件将更容易、更便宜、更机动、更可靠，通常也比桌面软件更强大。

使用互联网软件，除了软件本身，大多数用户不需要知道别的事情。所有那些乱七八糟、经常变动的东西，都放在服务器端，由精通此道的专业人员维护。所以，大多数时候，你不需要一台全功能的电脑（即本身可以运行软件的电脑）。你所需要的设备只要有键盘、屏幕、浏览器就够了，可能还有无线网卡。这样的设备没准就是指手机。不管它是什么，

它肯定是一种消费类电子产品，价格大概在 200 美元左右，所以人们主要根据外观选择购买。你支付的上网费会超过硬件费用，就好像现在你的电话费超过电话机的价格一样。[①]

数据在客户端与服务器之间走一个来回大概耗时 0.1 秒，所以与用户密集互动的软件（比如 Photoshop）仍然会把数据处理的部分放在桌面端。但是看看大多数人使用计算机的目的，你会发现 0.1 秒的时滞根本不成问题。我妈妈其实真的不需要一台桌面电脑，完全可以用互联网软件替代。像她这样的电脑用户有很多。

用户的胜利

我家附近，一辆汽车的保险杠贴着一张粘纸，上面写着“太麻烦，不如死”（death before inconvenience）。大多数人，在大多数时候，总是选择最省事的做法。如果互联网软件能够击败桌面软件，一定是赢在更方便这一优势上。无论从用户的角度还是从开发者的角度来看都是如此。



使用那些纯粹的互联网软件，你只需要一个能够上网的浏览器即可。所以，它不受地域限制，在任何地方都可以使用。但是，如果你使用安装在计算机上的桌面软件，那么就只能在这台计算机上使用。更糟的是，你的文件也存在这台计算机上。随着互联网越来越深入人心，桌面模式的弊端也就越来越明显。

最典型的例子就是网络界面的 Email。大家现在都认定，随时随地都应该可以收发 Email。如果 Email 是这样，为什么日程表不能这样呢？如果你能看到同事的文档，为什么不能编辑它呢？为什么你的数据非得禁锢在一台遥远写字桌上的电脑里呢？

---

① 有些公司生产轻量级的消费类电子产品，它们意识到“服务才赚钱”以后，往往就会把在线服务与硬件捆绑在一起卖。这个模式效果不好，第一个原因是，消费类电子产品和在线服务是两种类型的业务，需要两种不同的公司来做；第二个原因是，消费者不喜欢硬件和服务捆绑在一起收费。吉列公司赠送刀架，只靠刀片赚钱，这个模式可能只对吉列公司有效，而且剃须刀是低值易耗品，对用户承担的责任远远小于一个可以上网的终端设备。

手机制造商就很满足于只卖硬件，不捆绑服务，不试图分享电信公司的收入。这个模式应该也是互联网终端设备的商业模式。如果某家公司生产出一个外观精美的小型设备，里面包含了浏览器，可以通过任何ISP连接上网，那么全国的技术爱好者都会愿意购买的。

“你的电脑”这个概念正慢慢成为过去时，取而代之的是“你的数据”。你应该可以从任何电脑上获取你的数据。或者更准确地说，在任何终端设备上获取你的数据，终端设备不一定是电脑。

终端设备不应该存储数据，它们应该像电话那样。事实上，终端设备最后可能就会变成电话，或者反过来，电话变成终端设备。终端越做越小后，你可以每天把它带在身上，就更没理由把数据存储在上面了，万一遗失或者被窃就很麻烦。把 PDA 遗忘在出租车上无异于损失一块硬盘，唯一的区别是你的数据现在掌握在别人手里，而不是被擦掉了。

有了互联网软件，你的数据和软件本身都不保存在终端设备上，不用安装就能使用。既然不用安装，也就不用担心安装出错了，再也不会有应用软件与操作系统不兼容的问题了，因为软件与你使用的操作系统彻底无关。



由于没有安装这一步，所以在“购买”之前试用互联网软件将变得非常普遍、非常容易。只要联上网站，应该就能免费试用该网站提供的服务。Viaweb 的整个网站处处都是鼓励用户试用的提示。

试用 demo 之后，就可以登记成为正式用户了，只需要填一个很简单的表单。这应该是用户需要做的最后一点“多余的事情”了。使用互联网软件，你不需要为新版本付出额外的费用，或者做额外的准备工作，甚至可能你都不知道软件已经升级了。

现在，升级不再对用户形成大的冲击。久而久之，软件变得更强大了。这需要开发者付出一定的努力。他们必须正确地设计软件，使得它能够平滑升级，不让使用者感到困惑。这就是互联网软件面临的新问题，不过解决办法是有的。

所有用户都使用同样版本的互联网软件，bug一发现就会立刻得到纠正。所以，它的bug应该比桌面软件少得多。在Viaweb，我记得未解决的bug最多一次也总共只有十个，大部分问题都是一发现就得到了解决，不会遗留下来。这要比桌面软件小一个或几个数量级。

互联网应用程序能够同时被多人使用，所以非常适合团队协作性的工作。大多数用户现在还不了解软件协同办公，否则估计他们会强烈要求大部分应用程序都具备这个功能。举例来说，允许两个用户同时编辑

一个文档是一项很有用的功能。Viaweb 允许多个用户同时制作一个站点，主要原因倒不是因为用户要求，而是考虑到这是开发软件的正确方式，但是最后发现大多数用户都希望这样。

如果使用互联网软件，数据会更安全。即使硬盘损坏的风险依然存在，但是与用户没有关系，他们可以从此不关心这件事。风险发生在机房。互联网软件的运营方会备份数据，不仅因为它们的系统管理员很关心这一类事情，还因为一旦数据丢失，公司将面临极大的麻烦。如果用户自己的硬盘坏了，他们不会发狂，因为不能去责怪别人；如果一家公司丢失了他们的数据，他们会怀着超乎寻常的怒火，冲着这家公司发飙。

最后，互联网软件不太容易感染病毒。如果客户端只运行一个浏览器，病毒运行的概率就比较小，本机的数据不会遭到破坏。而专门攻击服务器端的病毒比较容易防御。[①]

对于用户来说，使用互联网软件不会有太大的压力。我觉得，大多数 Windows 用户使用桌面软件的时候都感到紧张，会有相当大的心理压力。释放这种压力，对你的产品将是一种巨大的推动。



代码之城

对于开发者来说，互联网软件与桌面软件最显著的区别就是，前者不是一个单独的代码块。它是许多不同种类程序的集合，而不是一个单独的巨大的二进制文件。设计桌面软件就像设计一幢大楼，而设计互联网软件就像设计一座城市：你不仅需要设计建筑物，还要设计道路、路标、公用设施、警察局、消防队，并且制定城市发展规划和紧急事件的应对方案。

Viaweb 的软件包括：与用户直接对话的一些大型应用程序、被这些大型程序使用的程序、常驻后台报告系统出错的程序、重新启动出错部分的程序、生成统计报告或数据库索引的程序（偶然运行）、回收资源或者移动及恢复数据的程序（手动运行）、伪装成用户的程序（为了测试系

① 安全问题的关键是不要有漏洞，而不是任何设计上的决策。服务器软件的性质决定了开发者对漏洞会加倍注意。而且，服务器被入侵会使得运营公司遭受巨大损失，所以它们为了在行业中生存下去，可能也会对安全问题备加关注。

统和发现 bug)、诊断网络故障的程序、完成备份的程序、对外提供服务界面的程序、实时显示服务器状态和访问数据的程序（很受用户欢迎，对我们也是必不可少的)、修改后的开源软件程序（包括修正 bug）以及许许多多的配置和设定文件。在我们被雅虎买下以后，Trevor Blackwell 写过一个令人叹为观止的程序，可以在不关闭网站的情况下，将网上商店转移到另一个机房的服务器上。此外，还有向系统管理员发出传呼信号的程序，向用户发传真和电子邮件的程序，引导完成信用卡交易的程序，通过套接字、通信管路、HTTP 请求、SSH、UDP 数据包、共享内存、文件互相通信的程序。一部分 Viaweb 服务器上故意不安装某些程序，因为保证 Unix 系统安全的关键之一就是不运行那些不需要的东西，降低服务器被侵入的可能性。

光有软件还不够，我们还花了许多时间琢磨服务器应该如何配置。我们自己搭服务器，不仅为了省钱，也是为了让机器完全满足我们的需要。我们还考虑哪些 ISP 连接主干网的带宽比较大。我们一直与 RAID 供应商保持联系。



但是，硬件需要考虑的地方，不仅仅在于怎么才能避免出问题，还在于怎样才能最大地发挥它们的作用。只要你控制了硬件，就能为用户提供更多的功能。如果你的产品是桌面软件，你就只能规定硬件的最低配置，无法为了某一个功能而要求用户增加硬件。但是，如果你控制了服务器，你就能轻而易举地增加功能，使用户可以发出寻呼、发送传真、通过电话操作网站、使用信用卡付款等。你所需要做的只是安装相关的硬件。我们总是在寻找通过硬件增加新功能的方法，因为这可以赢得用户，还可以让我们超越那些不直接控制硬件的竞争者（他们要么出售桌面软件，要么通过 ISP 分销互联网软件)。

由于互联网应用程序由多种软件而不是单独一个二进制文件构成，所以可以使用多种编程语言开发。如果你的产品是桌面软件，一般来说，你总是被迫采用与操作系统一致的语言，也就是 C 和 C++。所以，这些语言就被认为是“正统的”软件开发语言（非技术人员尤其可能这样想，比如经理层和风险投资家)。但是，这其实是一个伪信号（artifact)，不能因为桌面软件是这样开发的，就认定所有软件都是这样开发的。对于

互联网软件，你可以使用任何你想用的语言。[1]当今，许多顶尖黑客使用的语言与 C 和 C++大相径庭：Perl，Python，甚至还有 Lisp。

对于互联网软件，没人规定只能使用某些语言开发，因为所有的硬件都控制在你手里，你想要用什么语言，就能用什么语言。不同的语言适合不同的任务，你应该根据不同场合，挑选最合适的工具。尤其是在竞争者存在的情况下，“可以这样做”就变成了“必须这样做”（详见后文），因为如果你不利用语言的优势，那就会听任对手超过你。

我们的大多数竞争者使用 C 和 C++，这使得他们的软件明显不如我们，因为（不考虑其他原因）他们无法解决 CGI 脚本不能识别用户状态的问题。如果用户想要修改表单的部分内容，你不得不把表单的所有内容都放在同一个页面上，然后在最下面放一个“更新”按钮。正如我将在第 12 章中解释的，通过使用 Lisp 这种许多人眼中的教学语言，我们使得 Viaweb 编辑器更接近桌面软件带给用户的体验。



软件的发布

互联网软件带来的最大变化之一，就是软件发布方式的改变。对于桌面软件来说，发布新版本是一个很痛苦的过程，整个公司不得不使尽全力，满头大汗地挤出一大块巨型代码。从过程和结果上来看，无异于一次分娩。

互联网软件则完全不同，就像你写给自己用的程序一样，修改起来很方便。软件的发布过程可以分解为一系列的渐进式修改，而不是猛地推出一个大幅变动的版本。常见的桌面软件可能一年发布一到两个新版本，而我们在 Viaweb 经常是一天发布三到五个版本。

一旦采用了这种新模式，你就会知道发布方式对软件开发的影响有多么重大。桌面软件开发之中的许多棘手问题，都是源自于它的那种灾难性的发布方式。

---

① 1995年我们创立Viaweb的时候，Java applet被认为是互联网软件的解决方案。但是我们觉得，applet采用的还是过时的概念，它还是要求下载软件到客户端运行。更好更简单的方法，应该是所有的工作都放在服务器端完成。我们在applet上面浪费了一点时间，还好没延误什么，但是数不清的其他创业公司经不起引诱，掉进了这个泥潭，它们几乎都没有逃脱失败的命运。

如果一年发布一个新版本，你很可能会以打包方式处理bug，把它们留着，然后一次性全部解决。在发布新版本前，你可能会修改和更换一半的代码，从而又引入无数新的bug。接着，质量监控人员（Quality Assurance）开始测试新代码，逐一列出新发现的bug，你再按照这张清单把它们一个个消除。通常没办法把清单全部做完，它随时都在增长，说实话，谁也不确定它到底会有多长。这就好像在足球场上捡小石块一样费劲，你永远不知道为什么软件内部会出这么多问题。最好的结果也不过是，你得到了一个统计学意义上“合格”的版本。

对于互联网软件来说，大部分的变化都是细微和渐进的，所以引入bug的机会比较小。而且，在发布前测试的时候，你知道应该最仔细地测试哪个部分——显然就是你修改过的部分。这使得你对代码的掌握变得牢固得多。一般来说，这时候你确实是对软件内部的情况一清二楚。当然，这不是说你把所有代码都装在了脑子里，而是说你阅读代码的时候，非常自如流畅，不会像侦探破案那样苦思冥想，而是像飞行员那样，瞄一眼仪表板，就对飞行状况胸有成竹。



桌面软件导致了bug的宿命论。你很清楚，发布出去的软件肯定有bug，你甚至早就准备好了应对机制（比如发布补丁）。既然如此，bug再多一点又何妨？没过多久，你要发布下一个版本了，你明知其中某个操作完全不能使用，但还是照样发布。苹果公司前几年就干过这种事。他们必须发布新版操作系统了，压力越来越大，发布日期已经推迟了四次，无法再推了，可是有些部分还一点儿没写（比如CD和DVD操作的部分）。怎么办？他们就把没写完的操作系统发布出去了，用户必须日后自己动手安装缺失的部分。

互联网软件的发布规则是：它运行不了，你就无法发布；一旦它能运行了，你就可以立刻发布。

这个行业的老手可能会想：你说得好听！软件运行不了，就不发布，但是如果你已经对外承诺了明确的发布日期，到时却没有准备好，怎么办？这个问题听起来有道理，但是事实上，你不会对互联网软件做出这样的承诺，因为它根本没有“版本”这个概念。你的软件是连续性渐变的，某些更新也许比较重大，但是“版本”这个概念不适用于互联网软件。

如果你还没忘记 Viaweb 的旧事，你可能会觉得我这么说听上去很奇怪，因为那时我们总是宣布将有新版本推出。这只是公关伎俩啦，我们知道媒体喜欢听到版本号。如果你发布一个大的版本更新（版本号的第一位数发生变动），它们就会以大篇幅报道；如果你发布一个小的版本更新（版本号小数点后发生变化），它们最多只用一段话提一下。

我们的一些竞争对手的产品是桌面软件，确实有版本号。对我们来说，这种发布方式只表明他们的落后，但是他们却因此把媒体的目光都吸引过去了。我们不想做局外人，所以也开始为自己的软件加上版本号。什么时候需要媒体宣传了，就开出一张单子，上面总结了自从上次“发布”以来，我们新增的所有功能，然后在上面填一个新的版本号，发出一个新闻稿，宣布新版本已经准备就绪了。真是神奇啊，从来没有人看穿我们的把戏。

到被收购的时候，我们已经这样干了三次，所以已经到了第四版。如果我没记错的话，那时是 4.1 版。Viaweb 变成 Yahoo! Store 以后，媒体的曝光就没有那么必要了。所以，虽然软件一直没有中断开发，但是版本号却悄悄地被放弃了。



软件 bug

互联网软件的另一个技术优势在于，你能再现大部分的bug。用户的数据都在你的硬盘上。如果某个用户使用软件时出错了，你就不必像开发桌面软件那样苦苦猜测到底发生了什么事情。一般来说，只要用户通过电话向你描述一番，你就能把问题再现出来。如果你的程序中有自动侦测错误的代码，那么不用等到用户找上门，你可能已经知道哪里出错了。

互联网软件每时每刻都在被使用。你的代码一上线，就会经历严酷考验。bug 很快就会浮出水面。

软件公司有时会受到指责，因为他们竟然把发现bug的任务交给用户去完成。说实话，我其实提倡这种做法。对于互联网软件，这样做的好处很多，因为它的bug相对比较少，而且处理周期比较短。我们连续不断

地发布新版本，所以bug就比较少。我们能够再现用户遇到的问题，又能在修复后立刻发布，使得用户不用等很长时间，所以大部分bug的处理周期都比较短。我们的bug数量一直不多，以至于没有必要使用一个正式的bug追踪系统。

当然，在发布之前，你应该对修改之处进行测试，避免出现重大的bug。难免会有一些bug成为漏网之鱼，不过它们纯属罕见情况下才会发生的个案，在真正接到用户投诉之前，几乎不会影响到什么人。只要你能立刻解决bug，对于普通用户来说，他们就会觉得你的软件几乎是毫无问题的。我觉得，普通的Viaweb用户可能一个bug都没遇到过。

解决新代码的bug要比解决历史遗留代码的bug容易。在自己刚刚写好的代码中，找出bug往往会比较快。有时，你只要看到出错提示，就知道问题出在哪里，甚至都不用看源码，因为潜意识中你已经在担心那个地方可能会出错。如果你要解决的bug出自于 6 个月前写好的代码（假定你一年发布一个新版本，那么 6 个月就是发现bug的平均时间），那么就麻烦了，就要大费周章了。那时，你对代码也已经不熟悉了，就更可能采用危险的方式解决问题，甚至引入更多的bug。[①]



早一点发现bug就不容易形成复合式bug，也就是互相影响的两个bug。举例来说，一个bug是楼梯很滑，另一个bug是扶手松了，那么只有当这两个bug互相作用时，才会导致你从楼梯上摔下来。在软件中，复合式bug是最难发现的bug，往往也会导致最大的损失。[②]传统的方法是："把软件彻底拆开，将所有bug统统清理干净。"这样做难免产生一大堆的复合式bug。如果软件是经常性发布，每次只有小幅度的变化，那么就不容易产生复合式bug。这就好比做扫除：你一直在打扫大厅，掉落在地板上的东西会被立刻清理，省得它们时间一长与其他东西粘在一起。

---

① 这个观点引申自特雷弗·布莱克韦尔（Trevor Blackwell）的话，他说："随着软件规模的增大，开发成本指数式上升。这可能是因为修正旧bug的原因。如果bug都能被快速发现，成本的上升形态就能基本保持线性。"

② 复合式bug有一个子类型：两个bug是互相弥补的，好比"负负得正"，软件反而能正常运行。这种bug可能才是最难发现的bug。当你修正了其中的一个bug，另一个bug才会暴露出来。这时对你来说，你会觉得刚才修正错了，因为那是你最后修改的地方，你就怀疑自己在那里做错了，但是你其实是对的。

有一种编程方法叫做“函数式编程”（functional programming），它对你会有帮助，可以避免一些副作用。函数式编程在学术文献中研究得比较多，在商业软件中用得比较少。但是，对于互联网软件，它却很有用。很难用纯粹的“函数式编程”完成整个程序，但是它可以用来编写一些重要的部分，使得这些部分易于调试，因为它们不包含“状态”（state），非常便于不断进行小幅的修改和测试。我大量使用这种方法开发 Viaweb 的编辑器，我们自己的脚本语言 RTML 就是一种纯粹的函数式编程语言。

桌面软件行业的人可能很难相信，找出Viaweb的bug几乎成了一种游戏。因为软件发布以后，大多数bug都是罕见情况下才会发生的个案，受到影响的用户往往都是高级使用者，他们喜欢试验那些不常用的、难度大的操作。高级使用者对bug的容忍度比较高，尤其如果这些bug是在开发新功能的过程中引入的，而这些新功能又正是他们所需要的，他们就更能理解了。事实上，因为bug不多，你只有经过一些复杂的过程以后才会遇到它们，所以高级使用者往往因为发现了bug感到很得意。他们打电话给客服时，多半是一副胜利者的口吻，而不是怒气冲冲的样子，好像他们击败我们得分了一样。



客户支持

当你可以再现错误时，你开展客服支持的方式就变了。大多数软件公司将客户支持看作提高客户满意度的一种方式。在这些公司看来，要么是客户打电话来，报告一个已知的bug，要么是客户执行了错误的操作，你必须判断出他到底什么地方做错了。这两种情况对公司的知识积累都没有太大益处。所以，你开始觉得客户支持是一件令人头痛的事情，决定将客服人员与开发人员尽可能分离。

Viaweb 不是这样。我们的客户支持是免费的，因为我们希望知道用户的反应。如果他们使用时遇到困难，我们希望立刻知道，这样就能再现错误、解决问题、发布新版本。

所以，Viaweb的开发人员总是与客服人员保持密切联系。客服人员坐在距离程序员只有 9 米的地方，知道自己可以随时打断程序员的工作，

提交新证实的bug的报告。遇到重大bug，我们就算在开董事会，也会马上回来修改程序。

我们的这种方法让所有人都感到满意。客户很高兴，拨打厂商服务热线是免费的，而且还被当作通风报信的人，受到郑重对待。客服人员也喜欢这样，因为这使得他们可以帮助用户，而不是对着用户读操作手册；程序员喜欢这样，因为他们能够再现bug，而不是通过模糊不清的二手报告了解bug。

我们的政策是当场修复bug，这改变了客服人员与黑客之间的关系。在大多数软件公司，客服人员是低工资的边缘人，黑客则是呼风唤雨的主宰者。这些公司有各种各样的bug报告流程，但是几乎都是单向式的：使用者打电话给客服人员报告bug，客服人员填写某种形式的表格，传递给程序员（可能会经质量监控部门之手），程序员把bug写入待解决问题的清单。Viaweb不是这样，在收到使用者的bug报告之后一分钟内，程序员就会对站在身边的客服人员说："没错，你是对的，这是一个bug。"客服人员从黑客嘴里听到"你是对的"，会感到欢欣鼓舞。客服人员告诉我们发现bug的时候，他们心里怀着期待，就好像小猫想让别人知道自己抓住了一只老鼠一样。这也使得客服人员在判断bug严重性时格外小心，因为这关系到他们的声誉。



我们被雅虎收购后，客服人员被移到离程序员很远的地方。直到那时，我们才意识到客户支持实际上就是质量监控，也是某种程度的市场营销。除了记录bug，客服人员还必须大概了解相关知识、回答与bug相关的一些问题、解释令使用者迷惑不解的功能等。[①]有时，他们也扮演了使用者的代理人，我们会问他们哪个新功能是用户更想要的，他们总是能做出正确的回答。

全身心投入

能够即时发布软件，对开发者是一个巨大的激励。步行上班途中，我经常会想哪些地方还需要变动，然后当天就予以实现。一些重大功能

① 我们在Viaweb举办过一个比赛，看谁能说出我们软件中最差劲的地方。两位客服人员并列第一，我至今想起他们的叙述都不寒而栗。比赛一结束，我们立刻修复了那两个问题。

也是这样来的。即使某个功能要花两个星期（或者更长时间）开发，我也很确定，一旦写完就可以立刻看到效果。

如果软件的新版本要等到一年后才能发布，我就会把大部分新构思束之高阁，至少过上一段时间再来考虑。但是，构思这种东西有一个特点，那就是它会导致更多的构思。你有没有注意过，坐下来写东西的时候，一半的构思是写作时产生的？软件也是这样。实现某个构思，会带来更多的构思。所以，将一个构思束之高阁，不仅意味着延迟它的实现，还意味着延迟所有在实现过程中激发的构思。事实上，将一个构思束之高阁，甚至会限制新构思的产生。因为你看一眼堆放在一边、还没有实现的构思，就会想“我已经为下一个版本准备了很多有待实现的新东西”，你就懒得再思考更多的新功能了。

大公司的做法不是立刻实现新功能，而是先对新功能做一个计划。我们在 Viaweb 就是因为这个原因而遇到了麻烦。投资者和分析家会问，你们对未来有何计划。真实的回答是，我们没有任何计划。我们有改进的想法，但是如果我们想到应该怎么改进，就已经把它实现了。接下来六个月我们要做什么？所有能想到的最佳改进。我不知道自己是否有胆量公开这么说，但这是实话。计划这个词，只是将构思束之高阁的另一种表达方式。只要想到好的构思，我们就立刻着手实现。



Viaweb 和其他许多软件公司一样，大部分代码都有明确的负责人，而且只有一个。如果你负责某件事，那就真的是你负责。除了你以外，没有人能批准（他们甚至都不知道）这部分代码的发布。如果你出错了，没有人会提醒你，唯一的代码保护机制就是你的羞耻心，你不想被同事当成傻瓜，这就足矣。我这么说或许会让人误以为 Viaweb 的代码是漫不经心地编写出来的。实际上我们的开发进度很快，但是把代码放到服务器上发布之前，我们会深思熟虑。提高软件可靠性的关键在于开发时全神贯注，而不是降低开发速度。正是因为飞行员全神贯注，他才能在夜间让一架 18 吨重的飞机以 225 公里的时速平安降落在航空母舰的甲板上，做得比小孩子切面包还要安全。

当然，这样写软件也有局限。它适用于小型的、由优秀可靠程序员组成的开发团队，不适用于大型的、充斥大量平庸之辈的软件公司，在

那里不是程序员想出好的构思，而是一个委员会集体批准坏的构思。

逆向的《人月神话》[①]

幸好开发互联网软件需要的程序员比较少。我曾在一家中等规模的桌面软件公司工作，那里的工程部规模就超过 100 人，但是其中只有 13 人负责产品开发，剩下的人负责软件发布、软件移植以及其他事情。开发互联网软件，你最多只需要那 13 个人，因为不存在软件发布、软件移植以及其他事情。

Viaweb 的开发者只有 3 个人[②]。我一直在不停地招聘，压力很大，因为我们要把公司卖掉。我们很清楚地知道，买家不愿花大价钱买下一个只有 3 个程序员的公司。（解决方法：雇更多的人，在公司内创设其他项目，让他们去做。）



开发软件需要的程序员人数减少，不仅意味着省下更多的钱。正如《人月神话》一书中所指出的，向一个项目增加人手，往往会拖慢项目进程。随着参与人数的增加，人与人之间需要的沟通呈现指数式[③]增长。人数越来越多，开会讨论各个部分如何协同工作所需的时间越来越长，无法预见的互相影响越多越大，产生的 bug 也越来越多。幸运的是，这个过程的逆向也成立：人数越来越少，软件开发的效率将指数式增长。我不记得我们在 Viaweb 开过讨论如何编程的会议。步行去吃午饭的路上，我们就能把该说的话说完，从来没有例外。

---

① 《人月神话》（*The Mythical Man-Month*）是布鲁克斯（Frederick Brooks）所写的一本软件项目管理名著。所谓“人月”就是一个人在一个月内所能完成的工作量。假如某个项目预估需要12个人月，那么派4个人处理这个项目，理论上需要3个月，派6个人则只需要2个月。但是，布鲁克斯认为这种换算机制在软件业行不通，是一个神话，因为软件项目是交互关系复杂的工作，需要大量的沟通成本，人力的增加会使沟通成本急剧上升，反而无法达到缩短工期的目的。在本质上，软件项目的人力与工期是无法互换的，当项目进度落后时，光靠增加人力到该项目中，并不会加快进度，反而有可能使进度更加延后。（该书英文版已由人民邮电出版社出版。）——译者注

② 罗伯特·莫里斯写了客户下单的前台订单系统，特雷弗·布莱克韦尔写了图片系统和后台管理系统（商家用来处理订单、查看统计数据、修改设置等），我写了站点生成器（商家用来搭建网站的外观）。订单系统和图片系统的开发语言是C和C++，后台管理系统主要是Perl，站点生成器是Common Lisp。

③ 这里的“指数式”（exponentially），我用的是口语中的表达方式。更合适的词应该是“多项式似的”（polynomially）。

如果要说有什么缺点，就是由于开发人员比较少，每个程序员都必须承担一点儿系统管理的责任。当你在服务器上发布软件时，必须有人监控服务器，但是由于人员太少，监控员只能由开发人员兼任。Viaweb有许多系统组件，变动非常频繁，导致应用软件和系统软件之间的界线很难区分。硬性指定一条界线将限制我们的开发。所以，虽然我们总是安慰自己，公司运营很快就能走上正轨，一两个月后就能平稳发展，那时就可以雇一个专职的系统管理员让他专门负责服务器了，但是这个愿望一直没有实现。

只要你还在很活跃地开发产品，就免不了要亲自做系统管理，我认为没有其他可能。如果你梦想写完代码，向服务器递交（check in），然后就可以回家，一天工作结束，这在互联网软件身上肯定没有实现的可能。互联网软件是活的，每时每刻都在你的服务器上运行。一个严重的bug影响的可能不是一个用户，而是所有用户。如果某个bug破坏了硬盘上的数据，更是必须马上修复，诸如此类。我们的心得是，第一年之后就不必每分钟都盯着服务器了，但是对新变动的部分一定要密切关注。不要在半夜里发布代码，然后回家睡觉。



关注用户

互联网软件不仅把开发者与他的代码更紧密地联系在了一起，而且把开发者与他的用户也更紧密联系在了一起。财务软件公司 Intuit 的销售方式很出名，他们的销售员会在软件零售店里向顾客做自我介绍，然后请求顾客跟他们回公司，以便进一步了解软件。如果你亲眼见到某人第一次使用你的软件，你就会知道软件的哪个地方最打动他。

软件应该做到用户认为它可以做到的事情。但是，你不知道用户到底怎么想，除非你亲眼看到他们如何使用你的软件，相信我，看到和看不到大不相同。互联网软件能够让你前所未有地了解用户行为。你不必再人为挑选一个小型的用户样本，进行重点观察。每个用户的每一次点击你都可以看到。所以，你不得不仔细斟酌到底要看哪些行为，因为你不应该侵犯用户的隐私。但是，即使是最常见的统计项目，也能提供大量信息。

因为你能得到用户数据，所以就不用依赖基准测试[①]了。基准测试不过是在模拟用户，而你现在能看到真实的用户。你想知道应该优化什么地方，那就登录到服务器，看看什么程序最消耗 CPU。你也会知道什么时候应该停止优化，当我们后来发现 Viaweb 编辑器的瓶颈是内存而不是 CPU 时，就知道可能应该停止优化了，因为我们没办法压缩用户数据的大小。（唉，其实是有办法的，但是做起来很不容易。）

效率对互联网软件至关重要，因为硬件费用由你支付。你的资本支出成本除以服务器所能支持的最大用户数量，就是你为每个用户付出的成本。如果你的软件效率高，你就能比同样硬件配置的竞争对手多发展用户，获得更多的利润。我们在 Viaweb 的时候，每个用户的硬件成本大约是 5 美元，现在应该更低，可能比把第一个月的账单寄给他们的成本还要低。如果软件效率足够高，每个用户的硬件成本现在可以接近免费。



关注用户不仅有助于优化程序，还有助于指导你的设计。Viaweb 的脚本语言 RTML 允许高级用户自定义页面风格。我们发现 RTML 有点像留言本，用户通过它向我们提建议，因为只有当预设的页面风格不能满足用户需求时，他们才会使用 RTML。举例来说，编辑器的工具栏原先是横跨页面的，但是许多使用 RTML 的用户将它放在左下方，于是我们也就把工具栏的默认位置调整为左下方了。

最后，通过关注用户，你可以知道他们在使用软件的过程中什么时候遇到了麻烦。因为顾客总是对的，所以这表明你需要修正软件。推广 Viaweb 的关键，就是允许访问者在线试用。这可不是仅仅展示几张图片，而是真的让你使用我们的产品。只需要五分钟，你就能自己搭建起一个真实可用的网上商店。

几乎所有我们的新客户都是通过在线试用发展起来的。我想大多数互联网软件都是如此。如果用户坚持从头到尾成功地完成在线试用，那么表明他们喜欢这个产品。如果他们感到很困惑或者很乏味，就不会坚持试用到底。所以，只要我们能让更多的访问者坚持完成在线试用，我

---

① 基准测试（benchmark test）指的是先设置一个基本的数据环境，测试应用程序的表现，然后把这个表现当作“基准”（benchmark），用来比较其他情况下应用程序的表现。——译者注

们的用户增长率就会提升。

我研究了用户点击行为，发现在某一个地方，在线试用的用户会停止前进，改为点击浏览器的“后退”按钮。（如果你写过互联网软件，你会发现“后退”按钮是设计中最费脑筋的问题之一，很有意思。）所以，我就在那个地方加了一条提示，告诉用户已经接近终点了，提醒他们不要点击“后退”按钮。这时，互联网软件的另一个好处就体现出来了，你做了修改，马上就可以得到反馈。完成在线试用的用户比例从 60%立刻上升到了 90%。由于新增付费用户的数量是完成在线试用的用户数量的一个函数，所以与修改前相比，我们的收入增长了将近 50%。

金钱问题

20 世纪 90 年代早期，我读过一篇文章，它称应该让用户像订报纸那样按照使用时间长短订购软件的使用权。第一眼看上去，这种说法好像很滑稽。但是后来，我意识到这个观点是对的，因为它反映了软件开发的现实：软件开发不是静态的，而是一个持续不断的动态过程。按照传统的软件销售模式，厂商每推出一个新版本，就会强迫现有用户重新出钱购买，然后安装升级，只有这样厂商才能持续不断获得收入。我认为，如果公开收取软件的订阅费，而不是让用户购买软件的所有权，操作起来会更自然、更简便。“订报纸模式”正是互联网软件天然的收费模式。



互联网软件不可能用自由软件的模式经营，只能由商业性公司来经营。因为经营互联网软件要承担很大的风险，会产生大量支出，没有人会免费做这件事情的。

对于软件公司来说，互联网软件是一个很理想的收入来源。你每个季度的销售额不是从零开始，而是拥有一个持续的现金流。因为互联网软件每时每刻都可以升级，所以你不用担心做错什么事。事实上，你不可能真的做错什么事，因为如果用户痛恨你对软件的修改，你马上就会知道。你也不会有坏账的烦恼，如果谁不付钱，你就停止对他的服务。此外，你也不可能遇到盗版问题。

没有盗版是一种“优势”，但也是一个问题。一定数量的盗版对软件公司是有好处的。不管你的软件定价多少，有些用户永远都不会购买。

如果这样的用户使用盗版，你并没有任何损失。事实上，你反而赚到了，因为你的软件现在多了一个用户，市场影响力就更大了一些，而这个用户可能毕业以后就会出钱购买你的软件。

只要有可能，商业性公司就会采用一种叫做“价格歧视”[①]（price discrimination）的定价方法，也就是针对不同的客户给出不同的报价，使得利润最大化。软件的定价特别适合采用价格歧视，因为软件的边际成本[②]接近于零。这就是为什么很多软件的Sun服务器版本比Intel服务器版本更贵的原因，因为如果一个公司购买Sun服务器，就表明它很有钱，不在乎对设备的投资，那么为什么不向它开个高价呢？盗版实质上是一种价格歧视，只不过针对的是最底层的消费者。我觉得，软件公司明白这个道理，所以故意对某些盗版行为睁一只眼闭一只眼。[③]由于互联网软件无法盗版，所以软件公司必须想出其他策略推广软件。



相比桌面软件，互联网软件卖得更好，因为它易于销售。你可能认为，购买商品时，人们是先做出决定，然后再购买，好像这个行为分成两个独立的步骤。创立Viaweb之前，我也是这样想的，不过我从未对这个问题进行过深入思考。事实上，第二步对第一步有反作用，如果某样商品购买起来很困难，人们就会改变主意，放弃购买。反过来也成立，如果某样东西易于购买，你就会多买一点。自从有了亚马逊网上书店，我买的新书比什么时候都多。互联网软件提供的服务可能是世界上购买起来最方便的东西，如果你试用完demo再买，那就更是如此了。购买时，除了输入信用卡号码以外，用户不应该再被要求做其他事。（要求用户做得越多，你担的风险就越大。）

软件公司有时会采用分销模式，让ISP分销互联网软件。这样做很不

① 价格歧视无所不在，所以当我发现1936年的《鲁宾逊—帕特曼法案》规定价格歧视在美国属于非法行为时，不免感到震惊。这个法案看来并没有得到严格执行。

② “边际成本”是一个经济学概念，指下一个单位产品的生产成本。软件的边际成本就是复制代码的成本，所以接近零。这意味着，对软件公司来说，增加一个用户几乎没有增加生产成本。它与价格歧视的关系在于，边际成本越低，厂商的定价空间就越大，它可以针对特定消费者定出很低的价格，从而达到扩大销售、利润最大化的目的。——译者注

③ Naomi Klein在*No Logo*一书中说，有些服装品牌的目标客户是“都市青少年”，这些品牌的专卖店对店内偷窃行为就睁一只眼闭一只眼，因为在它们的目标市场中，那些在店内行窃的“顾客”也是流行风尚的带头人，可能会带动本品牌的销售。

好。服务器必须在你自己的控制之中，因为你需要不断改进硬件和软件。如果你放弃对服务器的直接控制，你就放弃了互联网软件的大部分优势。

我们的几个竞争对手就采用分销模式，那是作茧自缚。我常常想，这可能因为他们受到某些西装革履、衣冠楚楚的家伙的压制。后者只看到分销管道的销售潜力是多么令人兴奋，却没有意识到这将毁了他们正在销售的产品。通过 ISP 分销互联网软件，就好比让自动售货机出售寿司。

目标客户

谁是互联网软件的目标客户？Viaweb一开始就把个人和小企业当作目标客户。我认为这是互联网软件的通行规则。这些客户决策比较灵活，又需要低成本的新技术，所以他们更愿意尝试新事物。

互联网软件往往也是大公司的最佳选择。（虽然大公司反应迟钝，不一定能意识到这一点。）最好的内部网（intranet）就是互联网（Internet）。如果一家公司采用互联网软件，那么这家公司的软件系统将表现得更出色，服务器将得到更好的管理，员工可以不受地域限制使用这个系统。

反对者往往声称互联网软件不安全。如果员工可以很容易地登录，那么坏人也可以很容易地登录。一些大型批发商就不太愿意使用Viaweb，觉得不能把客户的信用卡资料交给我们，而是放在自己的服务器上更安全。可能没法很婉转地向他们表达我们的观点，但是事实上，他们的服务器就是没我们的安全，我们对数据的保护几乎肯定比他们好。想想看，谁能雇到更高水平的网络安全专家，是一个所有业务就是管理服务器的技术型创业公司，还是一家服装零售商？不仅因为我们有技术水平更出色的员工，还因为我们比他们更关心数据的安全。如果一家服装零售商的服务器被入侵，最多只影响到这家公司本身，这件事也很可能在公司内部被掩盖起来，最严重的情况下可能还会有一个员工被解雇。但是，如果我们的服务器被入侵，就有成千上万家公司可能受到影响，这件事也许还会被当作新闻发表在业内新闻网站 CNet 上面，使得我们的生意做不下去，不得不关门歇业。

如果你想把钱藏在安全的地方，请问你是选择放在家中床垫下面，还是放在银行？这个比喻对服务器管理的方方面面都适用，不仅是安全性，



还包括正常运行时间、带宽、负载管理、备份等，都是我们占优。只有把这些事情都做对，我们才能保证自己生存下去。服务器管理对我们是生死攸关的大事，玩具制造商如何看待对人体不安全的玩具，或者食品制造商如何看待感染了沙门氏菌的食品，我们就如何看待有缺陷的服务器管理。

某种程度上，使用互联网软件的大公司就好像把它的IT部门外包出去了。虽然听起来很激进，但是我认为这样做很好。比起自己雇用系统管理员，外包可以让这些公司得到更好的服务。非IT公司的内部系统管理员没有行业竞争压力，日久天长就会变得工作效率低下、不负责任。要让员工表现优秀，必须有竞争压力。销售员必须面对消费者，程序员必须面对竞争对手的软件，但是内部系统管理员就像老年单身汉，能够激励他的外部压力几乎没有。[①]我们在Viaweb就有足够的外部压力，使得我们努力前进，不会被别人甩在后面。我们直接接到客户的电话，而不只是同事的电话。如果服务器被入侵，我们会急得跳起来。即使过了这么多年，我现在想起当时的情景，肾上腺素还是会迅速上升。



所以，互联网软件通常也是大公司的正确选择。可是，大公司太迟钝，不到最后一刻想不通这一点，就像以前他们迟迟没有发现桌面电脑才是计算机发展的方向一样。这也是大公司往往会购买很贵商品的原因之一，所以你高额投入，向大公司推销你的商品，也就变得值得了。

有钱的客户倾向于更贵的选择，即使便宜的选择更符合他们的需要，他们也不会买。这种现象普遍存在。原因就是，那些索要高价的人将更多的钱投入推销。Viaweb 不采用这种做法。互联网咨询公司从我们手里抢走了几个高端商家。他们说服这些商家，让他们相信更好的选择就是，花 50 万美元，将网上商店开在自己的服务器上。结果是意料之中的，当圣诞节购物高峰来临时，服务器的负载陡然上升，这些商家一个接一个地发现，他们的选择并不是那么正确。Viaweb 的系统远比大多数商家自

---

① 不少公司都很想知道，什么事情可以外包，什么事情不可以外包。一个可能的答案是，公司内部所有不直接感受到竞争压力的部门都应该外包出去，让它们暴露在竞争压力之下。（我这里所说的“外包”，指的是聘请另一个公司来执行，而不是指把业务部门转移到海外。）

己搭建的系统更高级更先进，但是我们付不起高额的宣传费，无法让他们明白这一点。我们的宣传费每月只有 300 美元，无法派遣一个衣冠楚楚、言之凿凿的团队到客户公司做演示。

有一段时间，我们构思了一种新类型的服务，名叫“Viaweb 黄金版”。它比我们普通类型的服务贵十倍，但是功能一模一样，唯一的区别就是有专人穿着西装面对面把它卖给你。我们从未把这个构思付诸实践，但是我很肯定，要是真推出的话，一定会有商家购买。

大公司付出的高价之中，很大一部分是商家为了让大公司买下这个商品而付出的费用。（如果国防部花 1000 美元买一个马桶座圈，部分原因是要让国防部买下它本身就需要花很多钱。）这就是为什么公司内部的局域网软件明明不可取、但是还会继续存在并且不断发展的一个原因。这样的软件更昂贵，但是你对这个难题就是无能为力。所以，最好的安排就是把个人和小企业客户放在第一位。其他的客户该来的时候就会来。



桌面电脑

在服务器上运行软件并不是新鲜事。事实上，旧有的模式就是这样，大型机的应用程序都是在服务器上运行的。如果这个模式真那么好，为什么以前没有获得成功呢？为什么随着桌面电脑的兴起，大型机变得黯然失色呢？

刚开始的时候，桌面电脑似乎对大型机不构成威胁。最早的桌面电脑用户只是一些黑客或者业余爱好者（那时，别人就是这么称呼黑客的）。他们喜欢微型计算机的原因只是价格便宜。有史以来第一次，个人拥有了自己的电脑。“个人电脑”（PC）这个词现在是日常语言的一部分，但是当它刚出现的时候，听上去简直就是痴心妄想，就像今天我们听到“个人卫星”（personal satellite）时的那种感觉。

为什么桌面电脑大获全胜？主要是因为它们的软件更出色。为什么它们的软件更出色？原因可能是这些软件是小公司写出来的。

我认为，许多人没有意识到最早的创业公司是多么脆弱和踌躇。许多创业公司的出现完全出于偶然。几个朋友在一起，白天都要上班或者上学，利用业余时间做出一个产品原型，如果这个东西看上去有市场，那么可能

就会开公司。在这个雏形阶段，任何重大的阻碍都会把公司扼杀在摇篮中。为大型机开发软件要求大量的前期投入。买一台开发用机已经很贵了，而且因为客户是大公司，所以你还需要一支看上去很气派的销售队伍，这样才能把软件卖出去。创立一家公司专门开发大型机软件是一项艰巨的任务，比你自己利用晚上的业余时间在苹果电脑上写一个东西艰巨得多。所以，只有很少几家为大型机开发应用程序的创业公司。

桌面电脑的出现，带动一大批新软件纷纷涌现，因为对于初生的创业公司来说，开发桌面软件更可行。桌面电脑本身相对便宜，客户又主要是个人，通过电脑商店或者邮寄就可以销售。

把桌面电脑推入主流市场的软件是 VisiCalc，它是第一个电子表格程序，是由两个人[①]在自家阁楼里写出来的，它能做到大型机都做不到的事。在那个时候，VisiCalc 太先进了，人们为了能够使用这个软件，纷纷去购买苹果电脑。这标志了一个新时代的开始，桌面电脑开始成为主流，因为许多创业公司为它写软件。



现在看上去，互联网软件也有这个趋势，因为也有许多创业公司在做这件事。电脑的价格越来越便宜，你可以同我们一样，用一台桌面电脑当作服务器，开始自己的创业。廉价芯片正在侵蚀工作站（你现在甚至很少听到这个词）的市场份额，并且占领了大部分的服务器市场。雅虎公司的服务器能够承受互联网上最高的工作负载，但是这些服务器的芯片与你的桌面电脑是一样的，都是Intel公司低价的处理器。一旦你写出自己的软件，只要搭建一个网站就能销售。几乎所有我们的用户都是听了他人的推荐或者看了媒体的报道。[②]

Viaweb 是一家典型的处于起步阶段的创业公司。我们对开公司这件

---

① VisiCalc的两个作者分别是丹·布里克林（Dan Bricklin）和罗伯特·弗兰克森。丹在几天之内就先用Basic语言写出了一个原型。接下来几年，他们一起合作（主要利用夜间）用6502机器语言写出了一个功能强大得多的版本。那时，丹在哈佛商学院读书，罗伯特名义上的正式工作是开发软件。“我们开公司其实没有太大风险。”罗伯特对我说，“倒闭就倒闭了，没什么大不了。”

② 这件事没我说的这么简单。口碑效应发挥效果需要很长一段时间。报道我们的媒体也不是很多，后来我们以每月1.6万美元（再加上一些认股权证）的代价，聘请了Schwartz Communications公司（他们大概是高科技行业中最好的公关公司），媒体报道才多了起来。但是，真正起到决定性作用的销售推广渠道只有一个，那就是我们自己的网站。

事都心存恐惧。开始的头几个月，我们安慰自己：就把这一切当作一个实验，随时都可以收拾摊子一走了事。幸运的是，除了一些技术问题，我们几乎没有遇到太大的障碍。写完软件，我们就把开发用的桌面电脑当作了服务器，插上电话线，就与外部世界连接在了一起。这个阶段，我们唯一的支出就是食品和房租。

现在，创业公司有更多的理由选择互联网软件创业，因为开发桌面软件越来越乏味了。如果你现在开发桌面软件，就不得不接受微软公司的授权条款，调用它的 API，为它那个 bug 百出的操作系统伤透脑筋。历尽了千辛万苦，你最终写出了一个大受欢迎的软件，这时你可能会发现，你所做的一切其实只是在为微软公司做市场调查。

如果某家公司想要开发一个平台，吸引他人在此之上创业，这家公司必须使得黑客本人愿意使用这个平台。这意味着它必须是便宜的，而且有着良好的设计。苹果公司的 Mac 电脑自从一问世，就在黑客之中很流行，许多黑客为它写软件。[①]你在 Windows 身上就很少看到这种现象，因为黑客不喜欢使用 Windows。现在，善于写软件的那类人更喜欢使用 Linux 或者 FreeBSD 操作系统。



我想，要是为桌面电脑写软件，我们就不会成立创业公司了。因为桌面软件必须能运行在 Windows 上，要给 Windows 写软件就不得不使用它，可是我们对它并没有兴趣。有了互联网就可以绕过 Windows，直接在 Unix 系统上发布软件，用户通过浏览器使用。这种趋势将会迅猛发展，很像 20 年前 PC 刚刚诞生时的情景。

微软公司

回想桌面电脑刚刚出现的时候，人人都对巨无霸 IBM 心存敬畏。现在很难想象那种感觉，但是我对此记忆犹新。现在，令人敬畏的巨无霸

① 你可能会问，如果Mac电脑真那么出色，为什么后来它的市场表现不佳？还是那个老生常谈的原因：成本太高。微软公司把所有精力都集中在软件上面，所以很多厂商只要专攻硬件就可以了，把硬件成本降了下来。单单是微软的软件或者第三方厂商的硬件都不足以赢得市场优势，但是它们结合起来，就在个人电脑出现后一段关键时期中主导了市场。苹果公司同时做软件和硬件，所以成本上没有优势。（但是，苹果公司还没有失败，如果它能把iPod升级成手机，并且将网络浏览器包括在其中，那么微软公司就有大麻烦了。）（译者注：这段话写于2001年9月，苹果公司的iPhone手机已于2007年6月上市。）

是微软公司，我想它面对新技术的威胁不会像 IBM 那样疏忽大意，自以为高枕无忧，毕竟微软的成功就是利用了 IBM 的疏忽。

我在前文提到，我的母亲并不真的需要一台桌面电脑。大多数用户都是如此。这对微软是一个问题，它也知道这一点。如果软件都在服务器端运行，就没有人要用 Windows 了。微软该怎么办？它会利用对桌面电脑操作系统的垄断，阻止或者限制这种新一代的软件吗？

我预计微软会推出某种服务器和桌面电脑的混合产品，让它的桌面操作系统专门与由它控制的服务器协同工作。至少它也会把文件放在服务器上，需要的用户可以自己去下载。我觉得微软不会把运算都放到服务器端，让客户只要有浏览器就行了，它不会走得那么远，除非走投无路了。如果只需要一个带浏览器的终端设备就能完成所有工作，你就不需要微软了。要是微软不能控制终端设备，它就只剩下一条路，就是把用户推向它自己的互联网软件。



互联网对于微软来说，就像《一千零一夜》神话中被关在瓶子里的妖怪，我想微软会有一段很艰难的时期，会千方百计防止妖怪从瓶子里钻出来。未来会出现无数不同类型的终端设备，要想全部控制它们实在是太难了。如果微软专攻某些终端设备，那么竞争者可以为其他终端设备提供应用程序，从而获得击败微软的机会。[①]

在互联网软件的世界，微软并不会因为它在桌面软件世界的成功而自动得到一席之地。也许通过努力它能为自己争取到一个席位，但是我不认为它会主宰这个新的世界，取得桌面软件领域那样的地位。

---

① 一个优秀的开源浏览器会帮到互联网软件，防止它们被微软扼杀。体积小、反应快的浏览器，本身就是好东西，会鼓励厂商开发小型的互联网设备。

优秀的开源浏览器最大的优点则是，会推动HTTP和HTML继续向前发展（就像Perl起到的作用一样）。你还记得当年网景公司每推出一个浏览器的新版本，就会为HTML语言加上一些新功能吗？这样的好事为什么要停下来呢？

举例来说，如果互联网软件能够区分某一个链接是用户主动点击还是被动跟随前往，那将极有帮助。要做到这一点，只需要对HTTP协议做一点小小的加强就可以了，允许在一个请求中包括多个URL地址就能解决问题。此外，HTML语言如果能支持级联式菜单（cascading menu），也将很有帮助。

只要谁能写出一个新的Mosaic浏览器，他就能改变世界。现在这样做会不会太晚了？1998年，许多人认为再推出一个新的搜索引擎已经太晚了，互联网世界已经定型了。谷歌证明了这种看法是错误的。如果新事物真的有重大改进，那么它总是可以找到生存空间的。

除了微软自己，没有人能让微软遭受严重挫折。随着互联网软件的崛起，微软不仅要面对新的技术问题，还要面对它自己毫无根据、一厢情愿的旧思维。微软需要把它现有的商业模式拆除，建设一个新模式，我看不到它有正视这个问题的任何迹象。它一心一意地坚持桌面软件模式，固然把它带到现在的地位，但是现在开始将成为它继续前进的障碍。IBM 曾经有过同样的处境，它没有正确应对。在很晚的阶段，它才进入微机市场，并且三心二意没有倾注全力，因为大型机是 IBM 的主要利润来源，发展微机就等于扼杀这头金牛，所以它感到很纠结。微软也同样感到纠结，因为它想保住桌面软件。看来金牛也会成为沉重负担。

我并不是说，互联网软件世界不会产生主宰者。最终，可能会有这样一家公司诞生。但是，我想这需要相当相当长的时间，在此之前则是一段欣欣向荣的混战时期，正如微机诞生的早期。这是创业公司的黄金时代，小公司竞相争艳，做出很酷的产品，使得自己快速发展起来。



创业公司

典型的创业公司行动快速，看上去不是那么正式，只有很少几个人，资金也有限。这几个人勤奋工作，技术放大了他们的决策。如果他们赌赢了，那就是一场大胜利。

开发互联网软件的创业公司会把与创业有关的每一件事做到极致。只用更少的人、更少的钱，就可以把软件写出来，并且开始运作。你必须打破常规、快速行动，循规蹈矩不可能成功。你完全能够在只有三个人的情况下让产品开始运营，你们唯一的办公场所就是一间公寓，里面放着一台连着 ISP 的服务器。我们就是这样做的。

纵观创业公司的历史，你会发现它们变得越来越小，越来越快，越来越不像正规的企业。1960 年，所谓“开发软件”就是 IBM 公司的那种形式，满满一屋子的人，他们都戴着牛角质眼镜架，系着细细黑黑的领带，勤勉地埋头写代码，每人每天可以完成十行。到了 1980 年，“开发软件”变成了 8 到 10 人的一个小组，他们穿着牛仔裤上班，在 VT100 终端上打字。现在，“开发软件”则是两个人坐在客厅里，一人捧一台笔记本电脑。（牛仔裤如今已经不能算是不正式的服装了。）

创业公司的压力很大，不幸的是，这一点在互联网软件业也发挥到了极致。许多软件公司的开发者都有一段睡在桌子底下（或者类似经历）的日子，尤其是在初创期。令人惊恐的是，对于互联网软件来说，这样的日子没有尽头，什么都不足以阻止这种事情成为常态。对于桌面软件来说，睡桌子底下的经历经常可以告一段落，等到软件发布了，我们就都回家睡上一个星期。互联网软件永远没有收工的那一天，如果你愿意，可以一直干下去，每天忙上 16 个小时。而且，你能够做到这一点，意味着竞争者也能做到这一点，所以长时间工作变成了一种必需，不得不如此。因为你能做到，所以你必须做到。这简直就是逆向的帕金森定律[①]。

除了长期加班，更可怕的事情是沉重的压力。传统上，程序员和系统管理员有不同的工作职责。程序员关注bug，系统管理员关注系统的基础设施。程序员可能一整天都在伏案编写代码，然后到了某个时间，就下班回家，不再去想代码了。系统管理员则是永远都无法把工作抛到脑后，可能凌晨 4 点就会被叫到机房，不过好在大多数时候他们的工作都不是很复杂。互联网软件的出现使得这两种工作结合在一起，因此把它们各自不同的工作压力也合在一起。程序员变成了系统管理员，但是工作职责的范围却没有明确界定，使得工作压力陡然增加。



我们创立 Viaweb 时，一开始的六个月都在编写代码。与其他创业公司一样，在这个早期阶段我们的工作时间也是很长的。换作桌面软件公司，度过这个艰苦阶段之后，一切就会变得轻松了，但是当我们结束这个阶段、打开服务器迎接访问者时，才发现与后来的日子相比，第一阶段的编程简直像在度假。雅虎收购 Viaweb，我们第一位的收获当然是金钱，第二位的收获就是能够卸下这些沉重的责任，让一家更大的公司去承担。

桌面软件迫使用户变成系统管理员，互联网软件则是迫使程序员变

---

① 帕金森定律（Parkinson’s Law）是英国作家诺斯科特・帕金森（Cyril Northcote Parkinson，1909—1993）在1955~1958年的一组系列文章的总称。在这些文章中，帕金森讽刺了英国的官僚主义，总结了许多常见的官僚主义的表现形式。“帕金森定律”后来成为这些表现形式的代名词，它包括很多内容，其中有一条就是“因为你必须做到，所以你能够做到”。因此，本文作者称“因为你能够做到，所以你必须做到”是逆向的帕金森定律。——译者注

成系统管理员：用户的压力变小了，程序员的压力变大了。这未必是坏事。如果你的创业公司正在与一家大公司竞争，这一点就很有利。[①]互联网软件提供了一种天然的途径，使得你可以用较少的人力完成较多的工作，从而超过竞争对手。创业公司对这一点应该感到心满意足。

勉强够用的网页

有一件事可能会打消你通过互联网软件创业的念头，那就是网页作为用户界面，功能实在是太弱了。我承认，这确实是一个问题。我们真的想改造 HTML 语言和 HTTP 协议，对许多地方进行加强。不过必须指出，目前阶段的网页刚好能满足需要。

以第一代微型计算机作为类比。那些机器上的处理器，本来的生产目的是用在诸如交通信号灯这样的电子设备上的，而不是用作计算机 CPU。但是，像Altair[②]的设计者爱德华·罗伯茨[③]这样的人，意识到这些处理器刚好能满足需要。你把一块芯片、存储器（第一台Altair的内存是 256 字节）、前面板的指示灯和开关组合在一起，就有了一台可以运行的电脑。能够拥有自己的电脑是非常令人兴奋的事情，所以许多人想买一台，即使它的性能实际上非常低下。



网页刚诞生的时候，也不是为了用作应用程序的界面，它只是刚好能满足需要。对于相当一部分使用者来说，打开浏览器就能使用软件本

① 由于个人经历的关系，特雷弗·布莱克韦尔对这一点的认识可能比其他任何人都深刻。他写道："我会进一步说，由于互联网软件的程序员非常辛苦，所以会使得经济优势根本性地从大公司向创业公司转移。互联网软件要求的那种工作强度和付出，只有当公司是其本人所有时，程序员才愿意提供。软件公司可以雇到能干的人，让他们去干轻松的事情，也可以雇到不能干的人，让他们去干艰苦的事情，但是无法雇到非常能干的人，让他们去干非常艰苦的事情。因为互联网软件的创业不需要太多的资本，所以大公司可以与创业公司竞争的优势就所剩无几了。"

② Altair 指的是MITS公司在1975年推出的Altair 8800微型计算机，CPU为Intel 8080。它是第一批微型计算机中最成功的产品，被认为是最早的个人电脑，上市第一个月就卖出了几千台。它所使用的总线，后来成为微机总线的第一个标准S-100，而微软公司的第一个产品就是为Altair开发的编程语言Altair Basic。——译者注

③ Ed Roberts（1941—2010），美国工程师。1970年创立MITS公司，1975年设计出了历史性的产品——微型计算机Altair 8800，开创了个人电脑的时代，后来被称为"个人电脑之父"。1977年，他卖掉了MITS公司，来到佐治亚州乡下研究医学，最后成为小镇上的医生。——译者注

身，这已足够吸引人了，用户操作界面的丑陋和不方便并不是一个严重的问题。也许你使用HTML语言无法写出最美观的电子表格，但是你能写出一个可以供多人在不同地点、不同终端设备上同时使用的电子表格，而且不用安装任何软件，只要有浏览器就可以。你还可以写出数据实时更新，或者一旦满足某些条件就会自动发出传呼信号的电子表格。更重要的是，你能用它创造出各种各样甚至还没有名字也没有人想到过的新东西。VisiCalc可不是某个大型机软件的微机版，而是完完全全的一种崭新的软件。



图 5-1　*Popular Electronics* 杂志，1975 年 1 月号[①]

当然，互联网软件不一定非做成互联网软件，它也可以做成某种形式的桌面软件。但是，我很肯定，这不是好主意。你自然可以轻易地假设所有人都会安装你的桌面软件，这样就不用考虑很多麻烦问题了，没过多久，你马上就对这一点深信不疑了，所有人肯定真的都会安装你的

① 1975年1月号的*Popular Electronics*杂志的封面报道就是Altair 8800型计算机，封面上的文字是："突破性的项目。世界上第一台可与商用型号媲美的微型计算机。Altair 8800，为您节省超过1000美元。"这是历史上第一篇对微型计算机的新闻报道，影响极大，标志个人电脑时代的来临。当时，哈佛大学二年级的学生比尔·盖茨就是因为看到了这篇报道，才决定和保罗·艾伦一起为Altair开发Basic语言的解释器，并于1975年7月成立微软公司，因此这篇报道直接促成了微软公司的成立。——译者注

软件。要是他们不安装，你就惨了。

相比之下，互联网软件不需要做任何关于安装的假设，只要能上网的地方，它就能运行。这个优势已经很大了，随着各种上网设备的迅速发展，优势还会继续扩大。你的软件能用，又不必安装，用户就会喜欢你，你的日子也好过得多，因为你不会被各种各样桌面端的问题烦死。[①]

我感到，自己对互联网的观察不会输给其他任何人，但是我也无法预料网络终端设备会怎么发展。一个网页制作比较规范统一的时代也许会到来，但是在什么时候呢？

一切最终会变成什么样子？我不知道。如果你把赌注押在互联网软件上，你也不必担心这个问题。只要人们继续上网，互联网软件就输不了。互联网也许不是唯一的提供软件的途径，但是它现在就能发挥作用，并还将持续很长一段日子。互联网软件的开发成本低，即使是最小型的公司，也可以很容易地制作和发布。互联网软件做起来很辛苦，还有许多特别大的压力，但是这样只会使得创业公司成功的机会变大。



为什么不尝试一下？

E. B. 怀特[②]曾经从一个农民朋友那里听到一则趣闻。许多农场用电篱笆防止奶牛逃跑，但是不少电篱笆其实并没有通电。不过奶牛们已经吃过苦头，显然学会了不去碰电篱笆，这时不通电也能起到效果。“奶牛们，行动吧！”他写道，“趁着统治者打鼾时，夺回你们的自由！”

如果你是一个黑客，并且梦想自己创业，可能会有两件事情令你望而却步，不敢真正开始采取行动。一件是你不懂得管理企业，另一件是你害怕竞争。可是实际上，这两件事都是没有通电的电篱笆。

首先，管理企业其实很简单，只要记住两点就可以了：做出用户喜欢的产品，保证开支小于收入。只要做到这两点，你就会超过大多数创业公司。随着事业的发展，你自己就能琢磨出来其他的诀窍。

---

① 如果我是你，我甚至不会考虑使用JavaScript，Viaweb就没用。我在网上遇到的大部分JavaScript都是不必要的，其中很多都不能用。如果有一天你可以在手机（或者PDA，甚至烤面包机）上浏览网页，天知道这些设备会不会支持JavaScript。

② E. B. 怀特（E. B. White，1899—1985），美国当代著名作家、散文家，常年担任《纽约客》杂志的主要撰稿人，以优美的语体风格著称于世。——译者注

刚开始的时候，你可能入不敷出，但是只要亏损不持续太久，你就不会有事。如果初期阶段缺少资金，这至少有助于你养成勤俭节约的习惯。开支越小，就越不会超支。幸运的是，编写一个互联网软件是非常便宜的。我们的总支出就不超过 1 万美元，现在应该更便宜了。其中，我们不得不花了几千美元买了一台服务器，又花了更多钱购买 SSL。（那时唯一出售 SSL 软件的公司就是网景。）现在，你可以租到一台强大得多的服务器，上面已经安装好了 SSL，而费用比我们当时的带宽费还要少。如今，开发一个互联网软件的费用比购买一把高级办公椅还要便宜。

至于如何做出用户喜欢的产品，下面是一些通用规则。从制造简洁的产品开始着手，首先要保证你自己愿意使用。然后，迅速地做出1.0版，并且不断加以改进，整个过程中密切倾听用户的反馈。用户总是对的，但是不同的用户要求不一样。低端的用户要求简化操作和清晰易懂，高端的用户要求你增加新功能。软件最大的好处就是让一切变得简单。但是，做到这一点的方法是正确设置默认值，而不是限制用户的选择。如果竞争对手的产品很糟糕，你也不要自鸣得意。比较软件的标准应该是看对手的软件将来会有什么功能，而不是现在有什么功能。无论何时，你都要使用自己的软件。Viaweb 的主要功能是建立网上商店，但是我们也使用它建立自己的网站。不要只因为对方的头衔是市场专家、设计师或产品经理，就盲目听从他们的话。如果他们的观点真的很好，那就听从他们，关键是你要自己判断，不要盲从。只有懂得设计的黑客，才能设计软件，不能交给对软件一知半解的设计师。如果你不打算自己动手设计和开发，那就不要创业。



其次，让我们来看看竞争。你所害怕的大概不是与你一样的黑客，而是那些像模像样，有着办公室、商业计划、销售员的公司，对不对？可是实际上，他们害怕你胜过你害怕他们，而且这一点上，他们并没有错。几个黑客搞懂如何租用办公室，或者如何雇用销售人员，要比那些公司（不管大公司还是小公司）搞懂如何正确写出软件容易得多。我在这两种地方都待过，所以知道这些。Viaweb 被雅虎收购后，我突然发现自己在为一家大公司工作，那感觉就好像在齐腰深的水中艰难行走。

我无意贬低雅虎。它拥有一些很好的黑客和顶尖的管理人才。对于

一家大公司来说，它可谓是出类拔萃了。但是，它的生产效率仅仅相当于小型创业公司的十分之一。没有任何一家大公司能做得更好。微软公司的恐怖之处在于，它大到可以开发任何软件，就像一座能够行走的大山。



图 5-2 比尔・盖茨，1977 年

不要被微软吓到。你能做到它做不到的事情，正如它能做到你做不到的事情一样。开发互联网软件不需要得到任何人的许可，没有人能够阻止你。你不需要去申请许可证，不需要在零售店的货架上谋得一席之地，也不需要卑躬屈膝地求人家，将你的软件与操作系统捆绑在一起。你能够通过浏览器发布软件，没有人能在你和浏览网站的用户之间插上一脚。

你也许不会相信，但是我向你保证，微软公司害怕你。它的那些目中无人的中层管理人员也许不是这样想的，但是比尔・盖茨肯定是，因为 1975 年，上一次发布软件的新方式出现时，他也曾经跟你一样白手起家。



6

如何创造财富

如果你想致富，应该怎么做？我认为最好的办法就是自己创业，或者加入创业公司。几百年来，这一直是致富的可靠途径。“创业公司”（startup）这个词诞生于 20 世纪 60 年代，但是它与中世纪集资进行的航海冒险活动其实也相差无几。

创业公司往往与技术有关，所以“高技术创业公司”这个短语几乎就是同义重复。创业公司其实就是解决了某个技术难题的小公司。



许多人对此一无所知，但也发了财。这就好像你不用学习物理学也能成为一个出色的棒球投球手。但是，我认为理解这些原理，有助于你取得成功。为什么创业公司必须是小公司？当创业公司不断变大时，它是否不可避免地失去创新能力？为什么创业公司往往选择在新技术领域创业？为什么如此之多的创业公司在开发新药或计算机软件，而不是在卖玉米油或者洗衣粉？

一个命题

从经济学观点看，你可以把创业想象成一个压缩过程，你的所有工作年份被压缩成了短短几年。你不再是低强度地工作四十年，而是以极限强度工作四年。在高技术领域，这种压缩的回报尤其丰厚，工作效率越高，额外报酬就越高。

下面举一个简单的例子说明这个经济学命题。如果你是一个 20 多岁的优秀黑客，每年的薪水大约是 8 万美元。这意味着，平均来看，你必须每年至少为公司带来 8 万美元利润，这样才能保证公司没有亏钱。但是，你的真正工作时间其实可以是公司上班时间的 2 倍，如果你全神贯

注，每小时的产出可以提高 3 倍。[1]如果再把大公司里令人讨厌的中间管理层除去（他们经常以主管的身份妨碍你的工作），你的效率可以再提高 2 倍。还有一个可以提高效率的地方：你不用再完成强行指派给你的工作，尽可以根据自己的愿望，做出最能发挥你聪明才智的成果。假定这会把工作效率再增加三倍。将这些因子放在一起做乘法，你的工作效率将是在公司时的 36 倍。[2]如果一个优秀黑客在大公司里的身价是每年 8 万美元，那么一个勤奋工作、摆脱杂事干扰的聪明黑客，他的工作相当于每年新创造 300 万美元的价值。

这个计算是很粗糙的，有很大的误差。我不会争论这些数字是否准确，但是计算的根据是靠得住的。我没有说放大因子不多不少正好是 36，但肯定是大于 10 的，在个别情况下甚至高达 100。

如果你觉得一个程序员一年做出 300 万美元的利润不太可能，那么不要忘记，我们谈的是他在极限情况下可以创造多少利润。这时，他的休闲时间为 0，工作强度之大足以危害到健康。



创业公司不是变魔术。它们无法改变创造财富的法则，它们只是代表了财富创造曲线远端上的一点。这里有一个守恒定律：如果你想赚 100 万美元，就不得不忍受相当于 100 万美元的痛苦。比如，你终生为邮政局工作，省下每一分工资，那也是赚到 100 万美元的一种方法。可是，

① 往往只有在创业公司里，你才能得到一种宝贵的工作环境，就叫做“不受干扰”。不同的工作对“不受干扰”有不同的要求。文稿校对人员每15分钟被打断一次，工作效率也不会有太大损失。但是，黑客要求的“不受干扰”的时间是非常长的，有时你要用1个小时才刚刚把一个问题理清。所以，人事部突然打电话要你去填一张表格，会造成巨大的成本损失。

这就是为什么当你打扰黑客让他们从屏幕前扭过头回答问题时，他们会恶狠狠地盯着你的原因。他们大脑内部精心构建的精巧建筑，瞬间就崩溃了。

仅仅因为工作经常受到干扰，黑客就会无法应对高难度的项目。这就是为什么黑客往往在深夜工作的原因，也是黑客无法在小隔间里写出优秀软件的原因（除非在半夜）。

创业公司的一个巨大优势就是不会有任何人来打扰你。没有人事部，也没有表格，自然也就不会有人打电话要求你填表格。

② 那些大公司的执行官看到创业公司员工的生产率是本公司员工的20或30倍时，自然很想知道怎样才能让自己的手下也这样拼命工作。答案很简单，付钱就行了。

许多大公司的内部，平均主义泛滥。如果采用自由市场制度那样的机制，你的公司就可以变成一个很有效率的地方。

这里的假设是，如果每个员工按照他创造的财富获得报酬，那么整个公司的利润将最大化。

不难想象为邮政局工作 50 年是何等漫长的压力。创业公司将你所有的压力压缩到三四年。承受较大的压力通常会为你带来额外的报酬，但是你还是无法逃避基本的守恒定律。如果创业那么轻松，那么所有人就都去创业了。

运气的成分

每年 300 万美元在大多数人眼里是一笔大钱，但是在另一些人眼里却不值一提。300 万美元算什么？我的目标是成为比尔·盖茨那样的亿万富翁。

现在先不考虑比尔·盖茨，因为名人不适合用来举例子，媒体只报道那些最有钱的人，而他们往往属于特例。比尔·盖茨很聪明，有决断力，工作也很勤奋，但是单单这样还不足以让你成为他。你还需要非同一般的好运气。



任何公司的成功历程中，运气都是一个很大的随机因素。那些你在报纸上读到的成功人士固然很聪明，很努力，但是他们的运气也不坏。比尔·盖茨肯定既聪明又努力，但是微软公司碰巧是历史上最大商业错误之一——DOS 操作系统的授权协议——的受益者。毫无疑问，比尔·盖茨肯定竭尽全力才诱导 IBM 犯下这个大错，他也淋漓尽致地利用了对手的错误，但是只要 IBM 方面有一个人稍微有一点脑子，微软公司的历史就将完全不同。当时，与 IBM 相比，微软只是一家不起眼的小公司，一家小小的零件供应商罢了。如果 IBM 按照常理要求独家购买微软的产品，不许微软向第三方提供，微软也只能乖乖地签字。这对微软来说是一笔丢不得的大买卖，而 IBM 能够很容易地从其他公司搞到一个操作系统。

结果却是，IBM 尽全力帮助微软控制了个人电脑的标准。从那时起，微软只要不停地做出产品就可以了，它从来不用做出有商业风险的决策。微软只要抱着授权协议不放，快速地复制新产品就行了。

如果 IBM 没有犯下这个错误，微软依然会是一家成功的公司，但是不会膨胀得这么大这么快。比尔·盖茨依然将成为富翁，但是只会排在《福布斯 400 富豪榜》接近末尾的地方，与其他年纪相仿的人位置差不多。

致富的方法有许多种，本文只谈论其中的一种，也就是通过创造有价值的东西在市场上得到回报，从而致富。其他许多种的致富方法包括赌博、投机、婚姻、继承、偷窃、敲诈、诈骗、垄断、行贿、游说、造假、开矿等。获得最可观的财富往往会涉及其中的几种方法。

通过创造有价值的东西而致富，这种方法的优势不仅仅在于它是合法的（许多其他方法如今都是不合法的），还在于它更简单。你只需要做出别人需要的东西就可以了。

金钱不等于财富

创造有价值的东西就是创造财富。你最好先搞清楚什么是财富。财富与金钱并不是同义词。[①]财富存在的时间与人类历史一样长久，甚至更长久，事实上蚂蚁也拥有财富。金钱是一种历史相对较短的发明。

财富是最基本的东西。我们需要的东西就是财富，食品、服装、住房、汽车、生活用品、外出旅行等都是财富。即使你没有钱，你也能拥有财富。如果有一台魔法机器，能够按照你的命令变出汽车，为你洗衣做饭，提供其他你想要的东西，那么你就不需要钱了。要是你身处南极洲内陆，再多的钱对你也是无用的，因为没有东西可买，你真正需要的是财富。



财富才是你的目标，金钱不是。但是，如果财富真的这么重要，为什么大家都把挣钱挂在嘴边呢？部分原因是，金钱是财富的一种简便的表达方式：金钱有点像流动的财富，两者往往可以互相转化。但是，它们确实不是同样的东西，除非你打算伪造货币，否则使用“挣钱”这个词会不利于理解如何才能挣钱。[②]

金钱是专业化的副产品。在一个高度分工的社会，你需要的大部分

① 近代历史上，政府有时都搞不清楚金钱和财富的区别。亚当・斯密在《国富论》中提到，许多国家政府为了保住“财富”，禁止出口白银或者黄金。但是，黄金和白银实际上只是一种交换媒介，留住它们并不会让一个国家变得更富有。如果物质财富保持不变，金钱越多，导致的唯一结果就是物价越高。

② “挣钱”的英语是make money，字面意思就是制作金钱。在本文中，make money指的是创造财富，money指的是金钱，两者并不一样。所以，作者才会说，使用make money这个词，会不利于你理解如何才能make money。——译者注

产品无法自己制造。你需要土豆、铅笔、住房以及别的东西，你不得不让别人来提供。

那么，你怎样才能让别人去种土豆并把它提供给你呢？方法就是给出对方需要的东西作为回报。但是直接与他人进行物物交换，你能得到的东西有限。如果你本人制作小提琴，而附近的农民对它都不感兴趣，那么你怎样交换食品呢？

当社会分工越来越精细后，人们发现解决方法就是把贸易过程分为两步。不是直接用小提琴交换土豆，而是先用小提琴交换金钱（比如银子），然后再用金钱交换你需要的东西。金钱就是交换中介，它必须数量稀少，并且便于携带。历史上，充当金钱的最常见交换媒介就是贵金属，但是现在我们使用另一种东西充当交换媒介，那就是美元。只要政府保证美元能够流通，那么它不需要物理形式的存在，就能充当交换媒介。

交换媒介的优点是，它使得交易可以进行下去。缺点是，它往往模糊了交易的实质。人们觉得做生意就是为了挣钱，但是金钱其实只是一种中介，让大家可以更方便地获得自己想要的东西。大多数生意的目的是为了创造财富，做出人们真正需要的东西。[①]



大饼谬论

许多人从小就认定世界上的财富总额是固定不变的，这样想的人数量多得惊人。任何一个正常的家庭，在某个时刻所拥有的财富总是一个固定的值。但是，家庭的财富总额与世界的财富总额不是一回事。

---

①“财富”这个词有很多意思，有些并不是指物质财富。我不想做深入讨论，研究到底什么才是真正的财富。我这里指的只是一种特定的技术层面上的“财富”——人们用金钱向你交换的东西。这是一种很有趣、很值得研究的财富，因为它使得你免于饥饿，而且人们是否用金钱交换这种财富取决于他们，不取决于你。

当你开始做生意时，很容易陷入一种迷思，认为只要把东西做出来就会有人要。在互联网泡沫的那段日子，我遇到一位女士，她喜欢户外运动，所以开办了一个户外运动门户网站。如果你真的喜欢户外运动，你知道自己应该做什么生意吗？其中一种就是从出错的硬盘挽救数据。

两者有何关联？没有关联啦。我只是借此表达我的观点，就是说如果你想要创造财富（这里指的是狭义的财富，也就是使你免于饥饿的东西），那么你应该抱着特别怀疑的态度，去思考那些着眼于你自己感兴趣的东西的商业计划。对于自己感兴趣的东西，你会觉得它们很有价值，但是它们恰恰最不可能与他人眼中有价值的东西发生重合。

谈到财富总额的时候，财富经常被形容为一个大饼。政治家说：“你无法把饼做得更大。” 如果指的是某个家庭银行账户上的金钱数量或者政府某年的税收，这样说是对的。确实无法把饼做得更大，你分到的越多，别人分到的就越少。

小时候我就对这一点深信不疑：如果富人拿走了所有的钱，那么其他人就变得更穷了。许多成年人至今都是类似看法的信徒。每当有人提到 x%的人口占有了 y%的财富，他的言下之意往往就包含了这种错误的观点。如果你打算创业，那么不管你是否意识到了，你都是在着手推翻这种大饼谬论。

这里令人混淆的地方就是金钱有其抽象含义的一面。金钱不是财富，而只是我们用来转移财富所有权的东西。所以，虽然在某些特定的情况下（比如某个家庭当月的收入），你能用来与他人交换的金钱数量是固定不变的，但是大多数情况下，世界上可供交换的财富不是一个恒定不变的量。人类历史上的财富一直在不停地增长和毁灭(总体上看是净增长)。



假设你拥有一辆老爷车，你可以不去管它，在家中悠闲度日，也可以自己动手把它修葺一新。这样做的话，你就创造了财富。世界上因为多了一辆修葺一新的车，财富就变得更多了一点，对你尤其是如此。这可不是隐喻的用法，如果你把车卖了，你得到的卖车款就比以前更多。

通过修理一辆老爷车，你使得自己更富有。与此同时，你也并没有使得任何人更贫穷。所以，这里明显不是一个面积不变的大饼。事实上，当这样观察的时候，你会很好奇，为什么有人会觉得大饼的面积无法增大。[①]

孩子在不知不觉中就懂得了这个道理。如果一个小孩想赠送另一个小孩一件礼物，但是又没有钱，他就会自己动手做。只不过一般情况下小孩子的动手能力不足，相比店里买来的礼物，他们自己做出来的东西从外观上就显得比较粗糙。说实话，我们自己为父母做的松松垮垮的烟灰缸恐怕就很难在市场上卖出去。

---

① 如果在修理旧车的过程中，你对环境造成了一些微小的破坏，那么你可能使得每个人都变得更贫穷了一点。但是即使把环境的成本考虑在内，这依然不是一个零和游戏，依然存在财富的净增长。我们可以举出这样的例子，比如一台坏机器里有一个零件松了，你把零件拧紧，机器可以重新运作，那么你就没对环境造成任何破坏，并且创造了财富。

手工艺人

最可能明白财富能被创造出来的人就是那些善于制作东西的人，也就是手工艺人。他们做出来的东西直接放在商店里卖。但是，随着工业化时代的来临，手工艺人越来越少。目前还存在的最大的手工艺人群体就是程序员。

程序员坐在电脑前就能创造财富。优秀软件本身就是一件有价值的东西。这里不存在大规模的流水线制造业，所以不用担心问题被混淆。你输入的文字符号就是一件完整的制成品。如果某人坐在电脑前，写出了一个不那么糟糕的浏览器（顺便说一句，这是一件很值得做的事），世界就会变得富有得多。

公司就是许多人聚在一起创造财富的地方，能够制造更多人们需要的东西。当然，有些雇员（比如收发室和人事部的员工）并不直接参与制造过程，但是程序员不然。他们真正地面对产品，一行行地写代码把产品做出来。所以，在程序员看来，事情再明显不过，财富就是被做出来的，而不是某个想象出来的神秘人物分发的大饼。



另一件程序员看来显而易见的事情就是创造财富的速率存在巨大的差异。Viaweb 的一个程序员有着惊人的生产力，我记得看着他工作了整整一天，拿出来的产品估计使得公司的市场价值增加了几十万美元。一个优秀程序员连续工作几个星期可能可以创造价值 100 万美元的财富。同样的时间内，一个平庸的程序员不仅无法创造财富，甚至还可能减少财富（比如引入了 bug）。

这就是为什么如此之多的最优秀程序员都是自由主义者的原因。我们这个世界，你向下沉沦或者向上奋进都取决于你自己，不能把原因推给外界。许许多多不创造任何财富的人——比如本科生、记者和政客——一听到最富有的5%人口占有全社会一半以上的财富，往往会认定这是不公平的。一个有经验的程序员很可能也认为这是不公平的。因为最顶尖的5%的程序员写出了全世界99%的优秀软件。

创造出来的财富不一定非要通过出售实现价值。至少直到最近，科学家一直在把他们创造的财富真正地捐献给社会。青霉素的发现使得我

们所有人都变得更富有，因为从此我们死于细菌感染的可能性变小了。人们需要的东西就是财富，治愈疾病肯定就是人们需要的东西。黑客经常开发开源软件让所有人免费使用，以此把自己的工作捐献给社会。FreeBSD 操作系统使我变得更富有。我自己的电脑就在使用 FreeBSD，雅虎公司所有的服务器都是如此。

工作是什么

在工业化国家，一个人至少在二十多岁之前，总是从属于这样或那样的某个组织。经过这么多年，你已经习惯了自己属于这样一群人，早上全部起床，都来到同样几幢建筑物，开始做自己正常情况下没兴趣做的事情。这样的组织变成了你身份标志之一：姓名、年龄、头衔、组织名称。如果你要做自我介绍或者他人要描述你，结果无非就是，张三，10 岁，某某小学的学生，或者，张三，20 岁，某某大学的学生。



当张三从学校毕业后，他应该要找工作。找工作其实就是加入另一个组织。表面上，这个组织与大学很相像。你先挑选想去的公司，然后向它递交申请。如果它觉得你不错，你就能加入了。你早上起床，来到一个新的地点，也是几幢建筑物，开始做你正常情况下没兴趣做的事情。仅有的区别就是，上班的日子不如上学的日子有趣，但是有人付钱给你，而不是你付钱给学校。但是，上学和上班的相似之处要大于它们的不同。张三，20 岁，某某大学的学生，现在变成了，张三，22 岁，某某公司的程序员。

事实上，张三的生活比他意识到的发生了更大的变化。虽然公司和学校都是类似的社会组织，但是如果你深入观察现实，就会发现很大的区别。

公司一切行为的目的都是盈利，从而生存下去。创造财富是大多数公司盈利的手段。公司的业务高度专业化，掩盖了它们都是在创造财富的这种相似性，你不要觉得只有制造业公司在创造财富。财富的一个重要元素就是地理位置。还记得前文假设的那台魔法机器吗？它会变出汽车，为你洗衣做饭，等等。但是，如果这台机器在美国，而你在中亚的某个地方，那么它对你也没多大用处。我们说，财富就意味着人们需要

的东西，那么把商品送到顾客手中也是人们需要的。许许多多不生产物质商品的公司都是在如此创造财富。几乎所有情况下，公司的存在目的就是满足人们的某种需要。

当你为一家公司工作时，这也是你所做的事情。但是，公司内部的各种层级使得这一点有时不容易觉察到。你在公司内部所做的工作是与许多人一起合作完成的，你只是其中的一分子。你觉得自己是为公司的需要而工作，可能不会觉察到你其实是为了满足顾客的某种需求而工作。你的贡献也许不是直接性的，但是公司作为一个整体必须提供某种人们需要的东西，否则不可能盈利。如果公司一年付给你的薪水是 x 美元，那么总的来说，你为公司提供的劳动必须至少价值一年 x 美元，否则公司的支出就会大于收入，最后只好关门歇业。

一个大学毕业生总是想“我需要一份工作”，别人也是这么对他说的，好像变成某个组织的成员是一件多么重要的事情。更直接的表达方式应该是“你需要去做一些人们需要的东西”。即使不加入公司，你也能做到。公司不过是一群人在一起工作，共同做出某种人们需要的东西。真正重要的是做出人们需要的东西，而不是加入某个公司。[①]



对于大多数人来说，最好的选择可能是为某个现存的公司打工。但是，理解这种行为的真正含义对你没有什么坏处。工作就是在一个组织中，与许多人共同合作，做出某种人们需要的东西。

更努力地工作

大公司会使得每个员工的贡献平均化，这是一个问题。我觉得，大公司最大的困扰就是无法准确测量每个员工的贡献。大多数时候它只是在瞎猜。在大公司中，你只要一般性地努力工作，就能得到意料之中的薪水。你不能明显无能或懒惰，但是谁也没觉得你会把全部精力投入工作。

---

① 许多人20岁出头时感到非常困惑和压抑。大学生活很有趣，可是已经过去了，上班的日子为什么会差别这么大？不要搞糊涂了，你现在已经从顾客变成了仆人。在这种新生活中获得乐趣是可能的。不过，你首先需要入门，门口的牌子写着“闲人勿进”。这种转变是一种冲击，如果你不赶快意识到这一点，事情将变得更糟。

但是，现实是你在工作上投入的精力越多，就越能产生规模效应。在某些行业，那些真正拼命工作的员工能够创造出比普通员工多十倍甚至百倍的财富。比如，程序员全力开发一个崭新的软件，要比让他常规地维护和更新一个现有的软件能创造更多价值，这等于开辟了一个新的收入来源。

成立公司的目的不是奖励那些全部精力投入工作的员工。你不能对老板说，我打算十倍努力地工作，请你把我的薪水也增加十倍吧！因为公司已经假定你在全力工作了。但是，真正的问题实际上在于公司无法测量你的贡献。

销售员是一个例外。他们产生的收入很容易测量，他们的薪水往往是销售额的一个百分比。如果一个销售员想更努力地工作，他马上就可以这样做，并且自动按比例得到更多的报酬。

除了销售员以外，还有一个职位，大公司可以雇到顶级人才，那就是高级的管理职位。原因也是一样的，因为这个职位的贡献能够被测量，高级经理对整家公司的表现负责。普通员工的表现往往很难测量，所以也没人要求他们做出突出表现。高级经理就不一样了，他们像销售员一样，不得不用数字证明自己。一个表现糟糕的 CEO 是不能推托说自己已经尽了全力的。如果公司的表现不好，就是他的表现不好。

如果一家公司真正能够按照贡献付薪，它将取得巨大成功。许多雇员会更努力地工作。更重要的是，这样一家公司将吸引那些工作特别努力的人，从而超越竞争对手。

但公司不可能对每个人都像销售员那样付薪。销售员是单独工作的，大多数雇员则是集体工作。假设有一家公司制造某种消费品，工程师为它实现各种功能，设计师为它设计一个漂亮的外壳，营销人员让顾客相信这是值得拥有的商品。请问如何评价每个人对这个商品销售额的贡献？还有，上一代产品的工作人员为这个公司树立了质量可靠的形象，请问最新产品的销售额有多少应该归功于他们？根本没有办法把所有人的贡献一一分解清楚。如果你能读懂消费者心理，你会发现消费者把所有上面这些因素放在一起看待。

你想更努力地工作，但是你的工作与其他许多人的工作混杂在一起，



这样就产生了问题。在大公司中，个人的表现无法单独测量，公司里其他人会拖累你。

可测量性和可放大性

要致富，你需要两样东西：可测量性和可放大性。你的职位产生的业绩，应该是可测量的，否则你做得再多，也不会得到更多的报酬。此外，你还必须有可放大性，也就是说你做出的决定能够产生巨大的效应。

单单具备可测量性是不够的。比如，血汗工厂的工人报酬是按照计件制计算的，这是一个只有可测量性、没有可放大性的例子。你的表现可以被测量，并且据此得到回报，但是你没有决策的权力。你能做的唯一决策就是以多快的速度完成工作。即使你做到最快，回报可能也只增加一到二倍。



在电影中扮演主角就是一种同时具备可测量性和可放大性的工作。你的表现可以用电影的总收入测量，同时也决定了电影的成败，所以也就具备了可放大性。

CEO 也是一种同时具备可测量性和可放大性的工作。公司的表现就是 CEO 的表现，所以它具备可测量性。CEO 的决策决定了整个公司的方向，所以它具备可放大性。

我认为，任何一个通过自身努力而致富的个人，在他们身上应该都能同时发现可测量性和可放大性。我能想到的例子就有 CEO、电影明星、基金经理、专业运动员。有一个办法可以发现是否存在可放大性，那就是看失败的可能性。因为收入和风险是对称的，所以如果有巨大的获利可能，就必然存在巨大的失败可能。CEO、电影明星、基金经理、运动员的头顶都悬着一把宝剑，随时可能掉下来。一旦他们搞砸了，他们就完了。如果你有一个令你感到安全的工作，你是不会致富的，因为没有危险，就几乎等于没有可放大性。

但是，如果你想同时具备可测量性和可放大性，不一定非当上 CEO 或电影明星不可。你只需要成为某个攻克难题的小团体的一部分就可以了。

小团体 = 可测量性

就算你无法测量每个员工的贡献，但是你可以得到近似值，那就是测量小团队的贡献。

整家公司产生的收入是可以测量的，如果公司只有一个员工，那么就可以准确知道他的贡献了。所以，公司越小，你就越能准确估计每个人的贡献。一家健康的创业公司可能只有 10 个员工，那么影响收入的人员因子最多也只有 10。

因此，创业或加入一家创业公司最可能实现前文的情况，那就是你对老板说，我打算十倍努力地工作，请你把我的薪水也增加十倍吧！但是，有两个区别。第一个区别是你的要求并非向老板提出，而是直接向顾客提出（毕竟老板只是顾客的代理人）。第二个区别是你并非一个人完成这个任务，而是在一个小团体中与其他几个有同样抱负的人一起合作完成。



一般情况下，小型团队都由多人组成。只有表演或写作这样的特殊工作，你才会一个人单干。你最好找出色的人合作，因为他们的工作和你的一起平均计算。

大公司就像巨型的古罗马战舰，一千个划船手共同划桨，推动它前进。但是，两个因素使得它快不起来。一个因素是，每个划船手看不到自己更努力划桨有何不同；另一个因素是，一千人的团队使得任何个人的努力都被大大地平均化了。

如果你从一千人中随便挑出 10 个人，把他们放在一条小船上，他们很可能会划得更快。胡萝卜和大棒同时形成对他们的激励。身强力壮的划船手看到他个人对船的前进速度有显著影响，就会受到激励。如果有人偷懒，其他人很容易发现，并会对他提出抱怨。

如果你从大船上挑选出10个最优秀的划船手，把他们组成一个团队，这时，十人小船的优势才会真正显示出来。小团队带来的各种额外激励会在他们身上发挥得淋漓尽致。这里最重要的是你挑选出了最优秀的划船手，每个人都是一千人中排在最前面1%的顶尖高手。对他们来说，将自己的工作与其他高手的工作平均化要比与平庸之辈的工作平均化让人

满意多了。

这就是创业公司的真正意义。理想情况下，你与其他愿意更努力工作的人一起组成一个团队，共同谋取更高的回报（相比他们为大公司工作的情况）。因为创业公司的团队往往是自发形成的，许多有抱负的创始人彼此之间早就相识（至少听说过对方），所以他们对彼此贡献的评估要比一般的小团体更准确。创业公司不仅仅是十个人的团队，而且是十个同类人的团队。

乔布斯曾经说过，创业的成败取决于最早加入公司的那十个人。我基本同意这个观点，虽然我觉得真正决定成败的其实只是前五人。小团队的优势不在于它本身的小，而在于你可以选择成员。我们不需要小村庄的那种“小”，而需要全明星第一阵容的那种“小”。

团队越大，每个人的贡献就越接近于整体的平均值。所以，在不考虑其他因素的情况下，一个非常能干的人待在大公司里可能对他本人是一件很糟的事情，因为他的表现被其他不能干的人拖累了。当然，许多因素都会产生影响，比如这个人可能不太在乎回报，或者他更喜欢大公司的稳定。但是，一个非常能干而且在乎回报的人，通常在同类人组成的小团队中会有更出色的表现，自己也会感到更满意。



高科技 = 可放大性

创业公司为每个人提供了一条途径，同时获得可测量性和可放大性。因为创业公司是小团队，所以具备可测量性。因为创业公司通过发明新技术盈利，所以具备可放大性。

什么是技术？技术就是某种手段，就是我们做事的方式。如果你发现了一种做事的新方式，它的经济价值就取决于有多少人使用这种新方式。技术就是钓鱼的鱼竿，而不是那条鱼。这就是创业公司与餐馆或理发店的区别。餐馆煎鸡蛋，理发店剪头发，每次只能为一个顾客提供服务，但是如果你解决了一个热门的技术难题，别人都会使用你的解决方案。这就是可放大性。

回顾历史，大多数因为创造财富而发财的人都是通过开发新技术而实现的。你不可能通过煎鸡蛋或剪头发而致富，因为使用你的服务的人

是有限的。13 世纪，佛罗伦萨人发明了精纺布，那是当时的高科技产品，这种新技术造就了佛罗伦萨的繁荣。17 世纪，荷兰人掌握了造船术和航海知识，那也是当时的高科技，因此荷兰人主宰了欧洲前往远东的航线。

小团队天生就适合解决技术难题。技术的发展是非常快的，今天很有价值的技术，几年后可能就会丧失价值。小团队在如今这个时代可谓如鱼得水，因为他们不受官僚主义和繁琐管理制度的拖累。而且，技术的突破往往来自非常规的方法，小团队就较少受到常规方法的约束。

大公司也能开发出新技术，就是开发得比较慢而已。大公司的规模决定了它们无法快速行动，也无法测量并奖励表现优异的员工。所以在现实中，大公司开发出来的新技术只出现在那些需要大规模资本投入的领域，比如微处理器、电厂、大型民用飞机等，因为在这些领域内创业公司没有能力与之竞争。不过，即使在这些领域，大公司还是依仗创业公司提供零部件和构思。



生物科技类和软件类的创业公司很显然都是解决高难度技术问题的。我认为，就算看上去与技术无关的商业类公司，其实也是解决技术问题的。比如，麦当劳是快餐连锁集团，它的发展依靠的就是设计出了一个快餐服务体系，可以复制到全世界每一个角落。每一家麦当劳连锁店都必须严格遵守操作规定，这使得它就像软件一样运作。所以，麦当劳其实也符合“一次开发，普遍适用”的模式。沃尔玛也是如此，它的创始人 Sam Walton 并不是因为经营零售业而致富，而是因为设计出了一种新型商店。

选择公司要解决什么问题应该以问题的难度作为指引，而且此后的各种决策都应该以此为原则。Viaweb 的一个经验法则就是“更上一层楼”。假定你是一个手脚敏捷的小男孩，身后有一条壮硕的大狗正在追你。你跑到楼梯口，这时应该上楼还是下楼？我觉得应该上楼。如果下楼的话，大狗可能跑得跟你一样快。上楼的话，大狗的庞大身躯就将成为劣势。不错，跑上楼你会比较吃力，但是大狗会感到更吃力。

在实际操作中，这就意味着我们故意选择那些很困难的技术问题。假定软件有两个候选的新功能，它们创造的商业价值完全相同，那么我们总是选择较困难的那个功能。不是因为这个功能能带来更多的收入，

而是因为它比较难。我们很乐于迫使那些又大又慢的竞争对手跟着我们一起走进沼泽地。创业公司就像游击队一样，喜欢选择不易生存的深山老林作为根据地，政府的正规军无法追到那种地方。我还记得创业初期我们是多么筋疲力尽，整天都为一些可怕的技术难题绞尽脑汁。但是，我还是感到相当高兴，因为那些问题连我们都觉得这么困难，那么竞争对手就更会认为是不可能解决的。

这不仅是创业公司运作的好方法，更是创业公司的本质。风险投资商（VC）知道这个道理，为它起了一个名字——进入壁垒（barriers to entry）。如果你有一个新点子去找 VC，问他是否投资，他首先就会问你几个问题，其中之一就是其他人复制你的模式是否很困难。也就是说，你为竞争对手设置的壁垒有多高。[①] 你最好做出令人信服的解释，阐明你的技术难以复制的原因。否则一旦大公司看到了，它们就会做出自己的版本，再加上它们的品牌、资本、经销能力，一夜之间就把你的市场全部抢走。那时你就像来到开阔地带的游击队，会被正规军一举歼灭。



设置“进入壁垒”的方法之一就是申请专利。但是专利的保护程度可能不高。竞争对手通常能找到绕过专利的方法。如果找不到，它们可能就不找了，直接侵犯你的专利，等着你去起诉它们。大公司不害怕打官司，这对它们是家常便饭。它们很清楚，打官司的成本高昂又很费时。你听说过 Philo Farnsworth 这个人吗？他是电视机的发明者，可是没有人知道他，因为他的公司没有从电视机上面赚到钱。[②]赚到钱的公司是 RCA，发明电视机给 Philo Farnsworth 带来的后果就是一场长达 10 年的专利诉讼。

俗话说得好，最好的防御就是进攻。如果你开发出来的技术是竞争对手难于复制的，那就够了，你不需要依靠其他防御手段了。一开始就

① VC问我们，如果另一家创业公司开发与我们同样的软件，需要多少时间。我们当时的回答就是，可能没人能做到。我觉得，这样说使得我们听上去很幼稚，或者很像骗子。

② 技术的发明人往往很难确定，可以明确无误地确认只有一个发明人很难。所以根据这条规则，如果你知道某种东西的“发明人”（比如电话、流水线、飞机、电灯、晶体管），那是因为他的公司用这种发明赚到了钱，并且公司的公关人员尽力散布发明人的故事。如果你不知道谁发明了某种东西（汽车、电视、计算机、飞机引擎、激光），那是因为其他人的公司从这种发明中赚到了钱。

选择较难的问题，此后的各种决策都选择较难的那个选项。[①]

潜 规 则

如果创业就是比别人工作得更勤奋、赚到更多的钱，那么很显然人人都想去创业。而且一定程度上，创业也比较有趣。我觉得许多人都不喜欢大公司处事按部就班、会议没完没了、人际关系冰冷、管理层瞎指挥……

但创业是有一些潜规则的，其中一条就是很多事情由不得你。比如，你无法决定到底付出多少。你只想更勤奋工作 2 到 3 倍，从而得到相应的回报。但是，真正创业以后，你的竞争对手决定了你到底要有多辛苦，而他们做出的决定都是一样的：你能吃多少苦，我们就能吃多少苦。

另一条潜规则是，创业的付出与回报虽然总体上是成比例的，但是在个体上是不成比例的。我在前面说过，对于个人来说，付出与回报之间存在一个很随机的放大因子。你努力 30 倍，最后得到的回报在现实中并不是 30 倍，而是 0 到 1000 倍之间的一个随机数。假定所有创业者都努力 30 倍，最后他们得到的总体平均回报是 30 倍，但中位数却是 0。[②]大多数创业公司都以失败告终，其中并不都是很烂的项目（互联网泡沫时期曾经出现过专门介绍狗粮的门户网站）。一种很普遍的情况是，某个创业公司确实在开发一个很好的产品，但是开发时间太长了一点，结果资金都用完了，只好关门散伙。

创业公司不像能经受打击的黑熊，也不像有盔甲保护的螃蟹，而是像蚊子一样，不带有任何防御，就是为了达到一个目的而活着。蚊子唯一的防御就是，作为一个物种，它们的数量极多，但是作为个体，却极难生存。



① 总的来说，这也是很好的处事原则。如果你有两个选择，就选较难的那个。如果你要选择是坐在家里看电视，还是外出跑步，那就出去跑步吧。这个方法有效的原因可能是遇到两个一难一易的选择时，往往出于懒惰的缘故，你会选择较易的那个选项。在意识深处，你其实知道不懒惰的做法会带来更好的结果，这个方法只是迫使你接受这一点。

② 平均数（mean）是算术平均值，会受到个别极端值的影响；中位数（median）是最中间的那个值，不受个别极端值的影响。所以，这句话的意思就是，由于存在个别极其成功的创业者，所以回报的平均值被拉到了30倍，但是大多数创业者其实都以失败告终，所以中位数是0。——译者注

创业公司如同蚊子，往往只有两种结局，要么赢得一切，要么彻底消失。你通常不知道自己会是哪一个结局，只有等到最后一刻才会明了。有好几次 Viaweb 都接近失败了，我们的发展轨迹就像正弦函数的波形。幸运的是，我们在波形的最高点被收购了，但是真是差一点就倒闭了。在我们去加州拜访雅虎总部讨论公司出售事宜的同时，我们还不得不借了一间会议室专门安慰一位投资人，防止他撤出新一轮融资，因为没那笔钱我们就完蛋了。

创业公司这种大起大落的特点不是我们想要的。Viaweb 的黑客都是极度厌恶风险的人。如果有别的方式可以让努力与回报成正比，又不存在风险的因素，我们将很乐于尝试。我们宁愿以百分之百的把握去赚 100 万美元，也不愿以 20%的把握去赚 1000 万美元，尽管后者理论上的期望值比前者高出一倍。很不幸的是，如今的商界不存在百分之百把握赚到 100 万美元的可能。



保险的做法就是在早期卖掉自己的创业公司，放弃未来发展壮大（但风险也随之增大）的机会，只求数量较少但是更有把握的回报。我们曾经遇到过一个这样的机会，但是自以为是地将它放过了，事后才觉得自己很愚蠢。此后，我们就急不可耐地盼着把公司卖掉。在第二年里，只要有任何人对 Viaweb 流露出稍微一点点的兴趣，我们就试着努力把公司卖给他。但是，始终没有买家，所以我们不得不继续把公司开下去。

早期收购 Viaweb 是一桩很合算的交易，但是收购方对便宜货没有兴趣。一家大到有能力收购其他公司的公司必然也是一家大到变得很保守的公司，而这些公司内部负责收购的人又比其他人更保守，因为他们多半是从商学院毕业的，没有经历过公司的创业期。他们宁愿花大钱做更安全的选择，所以向他们出售一家已经成功的创业公司要比出售还处在早期阶段的创业公司更容易，即使会让他们付出多得多的价码。

用户数量

我认为，如果你的公司有机会被收购，那将是不错的选择。管理一家公司与创立一家公司是不同的两件事。当情况基本稳定下来以后，不妨让大公司来接手。这在财务上也是明智的选择，卖掉公司你的风险就

分散了，这就好像有一个理财师建议你用所有钱投资一支波动性很高的股票，你会怎么想?

那么，怎样才能把公司卖掉呢？基本上，不管是否想出售公司，你要做的事情都是一样的（比如多赚钱）。但是，被收购本身就是一门学问，我们在 Viaweb 花了很多时间研究它。

潜在的买家会尽可能地拖延收购。收购这件事最难的地方就是让买方真正拿出钱。大多数时候，促成买方掏钱的最好办法不是让买家看到有获利的可能，而是让他们感到失去机会的恐惧。对于买家来说，最强的收购动机就是看到竞争对手可能收购你。我们发现这会使得 CEO 们连夜行动。次强的动机则是让他们担心如果现在不买你，你的高速成长将使得未来的收购耗资巨大，甚至你本身可能变成一个他们的竞争对手。

在这两种收购动机中，归根结底的因素都是用户数量。你以为买家在收购前会做很多研究，搞清楚你的公司到底值多少钱，其实根本不是这么回事。他们真正在意的只是你拥有的用户数量。



事实上，买家假定用户知道谁有最好的技术。虽然这听上去很蠢，但是用户是你证明自己创造了财富的唯一证据。财富就是人们需要的东西，如果没人使用你的软件，可能不是因为你的推广活动很失败，而是因为你没有做出人们需要的东西。

风险投资商有一张清单，上面写满了各种表示不应该收购的危险信号。排在榜首的信号中有一个就是公司由技术顽童掌控，只想解决有趣的技术问题，不考虑用户的需要。你开办创业公司不是单纯地为了解决问题，而是为了解决那些*用户关心*的问题。

所以，我认为你应该和买家一样，也把用户数量当作一个测试指标。像优化软件一样优化公司，用户数量就是判断公司表现好坏的指标。做过软件优化的人都知道，优化难点就是如何测出系统的表现。如果凭空猜测软件最慢的是哪一部分以及怎样让它快起来，那估计百分百会猜错。

用户数量也许不是最好的测量指标，但应该也相差不远了。买家关心它，收入依赖它，竞争对手恐惧它，记者和潜在用户则是被它打动。无论你的技术水平有多高，用户数量都比你自己的判断更能准确反映哪些问题应该优先解决。

此外，将公司管理视同软件优化还能帮助你避免VC担心的另一个陷阱——开发某种产品的时间过长。现在，黑客都已经熟知这一点，并总结出一个术语“过早优化”（premature optimization）。尽快拿出 1.0 版，然后根据用户的反映而不是自己的猜测进行软件优化。

你必须时刻牢记的最基本的原则就是，创造人们需要的东西，也就是创造财富。如果你想通过创造财富使得自己致富，那么你必须知道人们需要什么。很少有公司真的关注如何使顾客更满意。有多少次，你走进一家商店，或者打电话给某个公司，你的心中怀着担忧和恐惧？当你听到“你的意见对我们很重要，请不要挂断”，你真的觉得事情会得到圆满解决吗？

餐馆有一道菜烧糊了，它还赔得起，因为只影响到一桌顾客。但是在科技行业中，你开发的新技术是供所有人使用的，一旦你的技术与使用者的需要有差距，影响就会被成倍放大。你要么令大量顾客满意，要么令大量顾客不满。你越能满足他们的需要，你创造的财富也就越多。



财富和权力

创造财富不是致富的唯一方法。在人类的历史长河中，它甚至不是最常见的方法。就在几个世纪前，财富的主要来源还是矿石、奴隶、农奴、土地、牲畜，而快速获得财富的方法只有继承、婚姻、征服、没收。所以，很自然地，财富的名声不好。

从那时到现在，两件事情出现了变化。第一个变化是法律。在相当长的历史时期内，你的财富得不到保护，统治者和他的手下可以设法将它占为己有。但是，变化出现在中世纪的欧洲。新兴的商人和制造业者开始在城市中崛起，[①]他们团结起来对抗当地的封建领主。人类历史上第一次出现强盗无法夺走平民血汗钱的情况。这对第二个变化起到了巨大的推动作用，甚至可能是第二个变化发生的主要原因。这第二个变化就

① 资产阶级在历史上首先出现在意大利北部和荷兰，这可能不是偶然，因为那里没有强大的中央政府。这两个地区是那时最富裕的地方，后来变成了文艺复兴向外扩散的两大中心。它们后来没能继续扮演这样的角色，那是因为其他地区（比如美国）将它们开创的模式发扬光大了。

是工业化的来临。

关于工业革命的起因，已经有大量的文献论述过。但是，创造财富的人能够心安理得地享用自己的财富，这确实是工业革命的一个必要条件（可能不是充分条件）。[①]一个反面证据就是，试图违背这个条件的国家经济都出现了倒退，比如 20 世纪六七十年代的英国工党政府（它的后果相对不太严重）。没有财富的激励，技术革新就会逐渐停顿。

还记得从经济学观点看什么是创业公司吗？简单说，就是可以让人更快速工作的地方。你不再是慢慢地积累 50 年的普通工资，而是要尽快地将这笔钱赚到手。所以，政府禁止个人积累财富实际上就是命令人民减慢工作的速度。他们同意让你在 50 年里赚到 300 万美元，但是不同意让你在 2 年里赚到这些钱，即使前提是你拼命努力工作。这样的政府就像一家大公司的老板，你无法对他说，我打算十倍努力地工作，请你把我的薪水也增加十倍吧！更严重的是，他永远是你的老板，即使你自己创业也避不开他。



缓慢工作的后果并不仅仅是延迟了技术革新，而且很可能会扼杀技术革新。只有在快速获得巨大利益的激励下，你才会去挑战那些困难的问题，否则你根本不愿意去碰它们。开发新技术是非常痛苦的经历，正如爱迪生所说，百分之一的灵感加上百分之九十九的汗水。没有财富的激励，就不会有人愿意去做技术革新。工程师愿意接受普通薪水去做一些诱人的项目（比如战斗机和登月火箭），而与日常生活关系更密切的技术革新（比如电灯泡和半导体）只能由创业者来发明。

创业公司并不只是过去二十年发生在硅谷的事情。如今，通过创造财富而致富已经成为了普遍的模式。每一个这样做的人差不多都应用了同样的诀窍：可测量性和可放大性。前者来自小团队的合作，后者来自开发新技术。无论是 13 世纪的佛罗伦萨，还是今天的加州，它们都是一样的。

理解这些有助于回答一个重要的问题：为什么欧洲在历史上变得如

---

① 充分条件在这里可能也成立。但是如果成立的话，为什么工业革命没有早一点发生呢？两个可能的（但是互相排斥的）解释是：(a) 变化其实早发生了，工业革命只是一系列革命中的一环；(b) 中世纪的城镇存在垄断经营和行会制度，延缓了新生产方式的诞生。

此强大？是因为欧洲优越的地理位置，还是因为欧洲人天生就比较优秀，或者是宗教原因？答案（或者至少是近因）可能就是欧洲人接受了一个威力巨大的新观点：允许赚到大钱的人保住自己的财富。

一旦自己的财产有了保证，那些想致富的人就会愿意去创造财富，而不是去偷窃。由此导致的新技术不仅被转化成财富，还被转化成军事力量。隐形飞机的理论是由前苏联数学家提出的，但是因为前苏联没有计算机工业，它就只能是一个理论，无法变成产品。前苏联没有足够快的硬件来完成设计飞机所需要的大量计算。

冷战、第二次世界大战、近代的大多数战争都说明了这个道理。要鼓励大家去创业。只要懂得藏富于民，国家就会变得强大。让书呆子保住他们的血汗钱，你就会无敌于天下。





7 关注贫富分化

当人们非常想把某件事做好的时候，有些人会做得比其他人好得多。达·芬奇的作品就比博格宁等同时代二流画家的作品优秀很多。同样的差距也存在于侦探小说家身上，雷蒙德·钱德勒的作品就比普通作家的作品好得多。顶级的国际象棋大师与普通的象棋俱乐部成员下一万盘棋，一盘都不会输。

与下棋、画画、写小说一样，赚钱也是一种专门的技能。但是，出于某种原因，我们以完全不同的态度对待这种技能。如果某些人善于下棋或写小说，没有人会有意见；但是，如果某些人善于赚钱，报纸上就会有社论出来说这是不对的。

为什么？赚钱看上去与其他技能没有本质不同，为什么人们的反应如此强烈？

我认为有三个原因使得我们对赚钱另眼相看。第一，我们从小被误导的对财富的看法；第二，历史上积累财富的方式大多名声不好；第三，担心收入差距拉大将对社会产生不利影响。就我所知，第一点是错的，第二点已经过时了，第三点通不过现实的检验。有没有可能，在现代社会中，收入差距拉大实际上是一种健康的信号？

财富的老爹模式

五岁时，我不知道电力是电厂生产的，以为插座就是发电的地方。同样，很多孩子以为财富是直接从父母口袋里流出来的，不知道财富是创造出来的。

由于孩子们接触到钱的方式就是这样的，他们往往会误解财富，把财富与钱混为一谈。他们认为财富的总量是不变的，某个权威负责分配



财富（所以理应平均分配），没有意识到财富是创造出来（而且创造得不太均等）。

事实上，财富与金钱是两个概念。金钱只是用来交易财富的一种手段，财富才是有价值的东西，我们购买的商品和服务都属于财富。你到海外旅行时，不用看当地人的银行账户就会知道你来到的是富国还是穷国。你只要看看他们的财富就行了：建筑、街道、服装、健康状况等。

财富从何而来？人类创造出来的。回到农业时代，这个概念就更容易理解。那时大多数人都务农，许多东西都需要自己生产出来。房屋、牲畜、谷仓等都是每个家庭自己生产出来的。这就很明显地说明，财富总量不是固定不变的，不像大饼那样会被分光。如果你想要更多的财富，自己生产就可以了。

这在今天的社会也成立，虽然已经很少有人直接创造财富供自己使用了（少量的家务活除外）。我们大多数人都在为其他人创造财富，然后用创造出来的财富交换金钱，再用金钱交换我们需要的另一种财富。[①]



孩子没有能力创造财富，他们享有的一切都来自别人无偿的给予。既然得到财富不要求对应的付出，那么它当然应该平均分配。[②]大多数家庭都是这样，如果兄弟姐妹中有人多得到了一份，其他孩子就会喊：“不公平！”

---

① 为什么财富的分配问题引起这么多争论？部分原因是反对声最大的人当中，很多人都少有创造财富的经验：大学生、继承人、教授、政客、新闻记者。（如果你在酒吧里听过大家议论体育赛事，你一定很熟悉这种现象。）

大学生往往依靠父母的资助，还没有想过父母的钱是从哪里来的；继承人靠着他人遗产过活；教授与政客距离创造财富最远，不管是否努力工作，得到的报酬都差不多；新闻记者部分由于他们的专业守则，必须与本行业产生收入的那部分——广告销售部——保持隔离。所有这些人中，有许多人从来没有直面过这个事实，那就是他们拿到手的金钱，都来自别人先前创造出来的财富（新闻记者除外，他们的工作是创造财富的，但是不直接用财富交换金钱）。在这些人的世界中，收入是由某个外部权威根据某种看似公平的抽象原则（对于继承人来说则是随机原则）进行分配，不是来自与他人交换别人需要的东西。所以，在这些人看来，真实世界的其他部分不采用同样的分配方式就是不公平的事情。

（某些教授确实为社会创造了大量财富，但是他们拿到的工资却不是对此的回报，更像是对他们的投资。）

② 如果你了解英国社会主义团体“费边社”（Fabian Society）的起源，你会觉得他们的想法好像是英王爱德华时代（1901~1910）的儿童读物《淘气鬼行善记》（*The Wouldbegoods*，作者Edith Nesbit）的小主人公想出来的，虽然看上去很无私高尚，但实际上非常天真幼稚。

进入社会以后，你不能总是靠父母养活。如果你需要什么东西，要么你自己做出来，要么做其他东西与需要的人交换金钱，再用金钱去买你想要的东西。在真实世界中，财富是你必须自己创造出来的东西（小偷和投机者除外），而不是等着老爹买给你。由于每个人创造财富的能力和欲望强烈程度都不一样，所以每个人创造财富的数量很不平等。

你做别人需要的东西或事情，然后得到报酬。有些人报酬较高，原因很简单，因为他们做得更好。大明星要比普通演员多赚许多钱，普通演员可能也有大明星的实力，但是人们在电影院选择看什么电影时总是被大明星吸引过去。

当然，做出人们需要的东西并不是赚钱的唯一方法。抢银行、索贿、垄断市场也能搞到钱，并且是某些富豪最大的财富来源，但是这些手段不能代表财富的全部，更不是贫富分化的主要原因。每个人的技能不同，导致收入不同，这才是贫富分化的主要原因，正如逻辑学的“奥卡姆剃刀”原则所说，简单的解释就是最好的解释。



在美国，一些大型上市公司的CEO的收入大概是普通人的100倍。[①]职业棒球选手的年收入是普通人的72倍，职业篮球选手的年收入则是普通人的128倍。报纸的社论用恐怖的语调引用这样的统计数据。但是我觉得，想象一个人的产出是另一个人的100倍是一件毫无困难的事情。在古罗马时代，根据奴隶的技能不同，他们的价格会相差50倍。[②]上述

① 根据Corporate Library的一项研究，2002年标准普尔500公司中，CEO总收入的中位数是365万美元（包括薪水、奖金、奖励的股票、执行后的股票期权）。根据《体育画报》的统计，2002至2003赛季NBA篮球选手的平均薪水是454万美元，2003赛季大联盟棒球选手的平均薪水是256万美元。根据劳工统计局的数据，2002年美国人的平均工资是35560美元。

② 在古罗马帝国的早期，一个普通成年男性奴隶的价格大约是2000赛斯特斯银币（参见Horace, Sat. ii.7.43），一个女仆的价格是600银币（参见Martial vi.66），而一个熟练园丁的价格是8000银币（参见Columella iii.3.8）。一位医生（P. Decimus Eros Merula）为了换取他的自由，付出了5万银币（参见Dessau, Inscriptiones 7812）。一位诗人（Calvisius Sabinus）出了10万银币购买懂得希腊文学的奴隶（参见Seneca, Ep. xxvii.7）。普林尼报告，在他的时代，奴隶的最高价格是70万银币，这位奴隶是语言学家（可能也是教师）Daphnis（参见Pliny, Hist. Nat. vii.39），但是这个价格后来被赎买自由的演员奴隶超越了。古希腊也出现了类似的价格分化。一个普通劳工的价格大约是125到150德拉克马银币，但是价格波动范围从50到6000银币（能够管理银矿的劳工）不等（参见Xenophon Mem. ii.5）。更多关于古代奴隶制的经济学研究参见Jones A. H. M.所著的“Slavery in the Ancient World”一文，发表于1956年的*Economic History Review*杂志的第185~199页。

收入差距还没有考虑激励因素或者现代科技带给你的生产力放大效应。

那些专门把运动员和 CEO 的收入拿出来说事的社论让我想到了中世纪的基督教教士。他们宁愿高谈阔论地球是不是圆的，也不愿亲自研究一下现实中的证据。[①]一个人的工作具有多少价值不是由政府决定的，而是由市场决定的。

“他一个人的价值真的等于我们 100 个人的价值吗？”社论作者这样问道。回答取决于你怎么定义“价值”。如果你同意“价值”可以定义为实现自身技能而得到的报酬，那么回答显然就是“对的”。

确实有一些 CEO 的收入太高，不合理，但是有没有 CEO 的收入不足以体现他所创造的财富的呢？乔布斯就是这样的例子。他拯救了濒临崩溃的苹果公司，扭转了危机，削减了成本，成功决策了下一代产品，很少有人能做到这些事情。他的收入就低于他的工作所创造的价值。如果我们不考虑 CEO 的例子，只说职业篮球运动员的收入，那么应该不会有太大争论，大家都会同意，篮球运动员的身价反映了市场供需状况，并没有不合理的地方。



第一眼看上去，你可能会觉得难以接受，人与人之间创造财富能力的差别真的会这么巨大吗？理解这一点的关键就在于重新思考我们上面提过的那个问题，他一个人的价值真的等于我们 100 个人的价值吗？你想一想，一个篮球队会同意用一个运动员交换 100 个普通人吗？如果苹果公司不是由乔布斯掌管，而是由一个 100 人组成的委员会掌管，那么这家公司的下一代产品会是什么样？[②]人与人之间的差别并不是那么稳定的线性关系。也许 CEO 和运动员的技能和决心只比普通人高出 10 倍（倍数不重要），但是人与人之间就是存在着重大差别。

当我们说一些工作报酬过高，另一些工作报酬过低，我们的真实想法到底是什么？在自由竞争的市场经济中，价格由买家的需求决定。如果人们喜欢棒球甚于喜欢诗歌，那么棒球运动员的收入就是要比诗人的

① 古希腊数学家埃拉托塞尼（Eratosthenes，公元前276—前195）通过测量不同地点建筑物的阴影长度，估算出了地球周长。他的结果只比正确值小了约2%。

② 我来告诉你这两个问题的答案。篮球队不会同意用运动员交换普通人，而苹果公司推出的产品就是Windows那个样。

收入高。如果说某种工作的报酬过低，那就相当于说人们的需求不正确。

当然，人们确实会需求不正确的东西。这有什么好奇怪的呢？你不觉得声称某种工作报酬过低的说法更奇怪吗？[①] 如果你觉得由于人们的需求不正确，导致某些工作的报酬过低而且不公平，那么这个世界一定会让你感到非常遗憾，人们就是喜欢看电视真人秀，而不是莎士比亚作品，人们就是喜欢吃玉米热狗，而不是水煮蔬菜，这是不是很不公平呀？要是你觉得不公平的话，那你就同把蓝颜色说成最漂亮的颜色、把方的说成圆的那样蛮不讲理。

“不公平”这三个字就是“老爹模式”的独门标志。为什么别的情况下人们不会想到这三个字？因为要是你现在还处于“老爹模式”，认为财富就是从某个口子流出来、被大家分享的东西，而不是来源于满足他人的需求的创造活动，那么当你注意到有些人赚钱比其他人多得多时，你就会不偏不倚地得出“不公平”这个结论。



当我们讨论“收入分配不公平”时，我们还要问问收入从何而来，[②] 收入背后的财富到底是谁生产出来的。如果收入完全根据个人创造的财富数量而分配，那么结果可能是不平均的，但是很难说是不公平的。

偷　　窃

很多人对贫富分化不满意的第二个原因就是，在大部分的人类历史中，积累财富最常见的方法其实是偷窃。游牧社会是偷别人的牲口，农

① 我们把由父母供给收入的模式称为“老爹模式”。这个模式与真实世界的最大区别之一就是对勤奋工作的评价不同。在老爹模式中，勤奋工作本身就是值得的，老爹会感到很高兴。但是在现实中，财富是用工作成果衡量的，而不是用它花费的成本衡量的。如果我用牙刷油漆房屋，屋主也不会付给我额外工资的。

所以，对于那些仍然处于“老爹模式”的人来说，看到有人勤奋工作却没有得到很多报酬就会感到不公平。为了破除这种迷思，让我们假设有一个工人，他单独一人在荒岛上打猎和采集水果。如果他的能力不足，就算非常勤奋地工作，最后也不会得到很多食物。这是不公平吗？又是谁对他不公平呢？

② 有那么多人相信“老爹模式”，部分原因是“分配”（distribution）这个词有双重含义。经济学家谈论“收入分配”（distribution of income）时，他们实际上指的是统计学上的收入分布。但是，如果你经常使用这个词，你会情不自禁将它与另一个意思联系起来（比如救济金的分配），因此下意识地就把财富看作从某个大水源流出来的东西。税收上，有一种税叫做“递减税”（regressive tax），其中“递减”（regressive）这个词也有类似的效果，至少我就是这样认为，一件东西是“递减”的，那么它怎么可能是好的呢？

业社会是征税（和平时期）和直接掠夺（战争时期）。

在战争中，胜利的一方将失败的一方的财产全部占为己有。1060年，征服者威廉占领英格兰，将当地贵族的财产全部分给他的随从，这是战争导致财富分配变化的一个例子。1530年，亨利八世将修道院的财产分给大臣[1]，这是政治斗争导致财富分配变化的一个例子。不管是战争还是政治斗争，本质上都属于偷窃。

在控制程度更高的社会，统治者和官僚阶层用税收代替直接充公。但是，根本的一点并没有变，那就是致富的方法不是创造财富，而是以统治者的强权进行搜刮掠夺。

随着欧洲中产阶级的崛起，这一切开始发生变化。按照我们现在的理解，中产阶级就是既不富裕也不贫穷的那部分人，但是在中世纪，中产阶级其实是一个独立的团体。封建社会只有两个阶级：贵族与农奴（为贵族服务的人）。中产阶级是一个新的第三类团体，他们出现在城镇中，以制造业和贸易为生。



从公元10世纪和11世纪开始，小贵族和获得自由的农奴聚集在城镇中，逐渐形成了与封建领主对抗的强大力量。[2]中产阶级主要通过创造财富谋生，这一点与农奴相同。（在热那亚和比萨这样的港口城市，中产阶级也会包括海盗。）但是，与农奴不同的是，中产阶级有强烈动机大量创造财富。农奴创造的所有财富都属于他的主人，所以大量创造财富对农奴来说意义不大。城镇的出现使得那里的人们可以独立生活，保住自己创造出来的财富。

一旦通过创造财富而使致富成为可能，社会从整体上就会快速地变得更富有。中世纪人们所需要的一切东西几乎都是由中产阶级生产出来

① 根据历史记载，“自从年轻的亨利八世登基，鲁斯勋爵（Thomas Lord Roos）就忠心耿耿地服侍左右，很快得到了回报。1525年，他被册封为嘉德骑士（Knight of the Garter），拉特兰（Rutland）成为他的领地。1530年后，他支持与罗马教廷对抗，积极镇压民间的宗教反抗，并且投票支持亨利八世以通奸罪审判皇后，将其处以死刑，这使得他成为接管修道院财产的有力候选人。”参见Lawrence Stone所著的*Family and Fortune: Studies in Aristocratic Finance in the Sixteenth and Seventeenth Centuries*一书第166页（1973年由牛津大学出版社出版）。

② 考古学证明确实存在大型的聚居地，但是当时的日常生活情况还是很难判断。参见Richard Hodges和Whitehouse David所著的 *Mohammed, Charlemagne and the Origins of Europe*一书（1983年由康奈尔大学出版社出版）。

的。工业革命后，其他两个阶级实际上消失了，他们的名称被用来指中产阶级的两端。（根据原始定义，比尔 • 盖茨不是富豪阶层，而是中产阶级。）

但是，创造财富真正取代掠夺和贪污成为致富的最佳方式，并不是发生在中世纪，而是发生在工业革命时代。至少在英国，当更快的致富方式出现后，贪污才逐渐不流行了（事实上，贪污从那时开始才被叫做“堕落”[①]）。

17 世纪的英国很像今天的第三世界，当官是公认的发财职业。那个年代要赚大钱仍然主要通过贪污，而不是经商。[②]到了 19 世纪，情况就变了，虽然存在大量贪污受贿（今天依然如此），但是政府逐渐被一些将良心和名誉看得比金钱更重要的人所控制。技术的发展使得通过创造而积累财富的速度第一次有可能超过通过偷窃而积累财富的速度。19 世纪典型的富人不是宫廷朝臣，而是实业家。



中产阶级的出现使得财富总量不再是一个固定不变的值，财富的分配也不再是一种零和游戏。苹果公司的两个创始人乔布斯和沃兹尼亚克没有使得他人变得更贫穷就赚到了钱。事实上，他们创造出来的东西使得人类的物质生活变得更富有。他们只能这样做，否则不会有人付钱给他们的。

即使情况已经发生变化，但是由于人类历史上主要的致富方式长期以来都是偷窃，所以我们依然对有钱人抱有一种怀疑态度。理想主义的大学生从小受到历史上知名作家的影响，长大后不知不觉保留了孩提时对财富的看法。这是一个双重误解的例子，就是对一个已经过时的情况持有错误的看法。

巴尔扎克说过：“每一笔巨大财富的背后，都隐藏着罪行。”这句话被广泛引用，但是他其实说的是另一个意思，如果巨大财富没有明显的

① “贪污”和“堕落”在英语中是同一个词：corruption。——译者注

② 16世纪的英国，权力最大的大臣分别是William Cecil和他儿子Robert Cecil。两人都利用职权获取了大量财富，成为当时最富有的人。Robert Cecil收受贿赂都到了叛国的地步。“身为国务卿和詹姆斯一世主要的外交政策顾问，他收受了很多好处。荷兰人向他大量行贿，要求英国不要与西班牙媾和，而西班牙人也向他大量行贿，要求英国与其签订和约。”

来源，那可能就来源于精心安排的犯罪活动，由于掩盖得太好，使得罪行被人遗忘了。如果我们正在谈论 11 世纪的欧洲，那么这样的误读反而是正确的。但是，巴尔扎克生活在 19 世纪的法国，那里的工业革命当时已经很发达了。巴尔扎克很清楚，你不用偷窃也可以发财。起码他自己就是这样做的，他写出受欢迎的小说，从而赚到了钱。[①]

技术的杠杆效应

技术的发展是否加剧了贫富分化？首先，技术肯定加剧了有技术者与无技术者之间的生产效率差异，毕竟这就是技术进步的目的。一个勤劳的农民使用拖拉机比使用马可以多耕六倍的田。但是，前提条件是他必须掌握如何使用新技术。

我自己就亲眼目睹过技术的这种杠杆效应不断扩大。高中时，我通过割草和在冰淇淋店当服务员赚钱，它们是我能找到的仅有的工作。现在的高中生可以通过开发软件或制作网站赚钱。不过，只有少数高中生具备这种能力，其余的人还是只能去冰淇淋店当服务员。



我清楚地记得，技术的进步使得我在 1985 年终于可以拥有一台自己的电脑了。只过了几个月，我就开始接一些编程的零活赚钱了。1985 年之前我就做不到这一点，那时也没有自由程序员这种工作。但是，苹果公司推出了强大而且便宜的个人电脑，使得一切成为可能，这本身就是在创造财富。程序员马上接了上去，使用苹果公司的产品，再去创造更多的财富。

正如这个例子所反映的，技术对生产效率的提高不是线性的，而可能是多项式形态（polynomial）的。所以，随着时间推移，我们应该会看到个人生产效率总是保持增长。这种增长会使得贫富差距不断扩大吗？这取决于你指的是什么“差距”。

技术应该会引起收入差距的扩大，但是似乎能缩小其他差距。一百年前，富人过着与普通人截然不同的生活。他们住在大房子里，有许多

---

① 虽然巴尔扎克从写作上赚到了很多钱，但他是出了名的挥霍无度，终生都受到债务困扰。

仆人服侍，穿着华丽但是不舒适的服装，乘着马车旅行（因此还有马厩和马夫）。现在，由于技术的发展，富人的生活与普通人的差距缩小了。

汽车就是一个很好的例子。如果富人不购买普通汽车，而是购买全手工制作、售价高达几十万美元一辆的豪华车，对他反而不利。因为对于汽车公司来说，生产那些销量很大的普通汽车要比生产那些销量很小的豪华车更有利可图，所以汽车公司会在普通车辆上投入更大的精力和资金，进行设计和制造。如果你购买专为你一个人定制的汽车，质量反而不可靠，某个部件肯定会出问题。这样做的唯一意义就是告诉别人你有能力这样做。

再来看手表的例子。50 年前，花巨资购买一块名表真的是很有面子的事情。那时的手表都是机械表，价格越贵，走时越准。现在不是这样了，石英表发明了，一块普通的石英表反而比几十万美元的名牌机械表走时更准。[①]说实话，就像汽车的例子一样，如果你一定要把钱花在手表上，结果只能给你带来更多麻烦：除了时间精度下降以外，机械表还必须上发条。



技术无法使其变得更便宜的唯一东西，就是品牌。这正是为什么我们现在越来越多地听到品牌的原因。富人与穷人之间生活差异的鸿沟正在缩小，品牌是这种差距的遗留物。但是，品牌只是商品的标签，即使买不起名牌，至少你还可以买普通牌子，这总比根本无法消费这一种商品要好得多。1900 年，只要你有一辆马车，你就是富人，根本没人问你马车的牌子。没有马车的人就是穷人，只能挤公共交通或者步行。今天，即使最穷的美国人也有自己的汽车，那么厂商只好通过广告训练我们识别品牌，以便我们能够识别哪些汽车特别昂贵。[②]

这种变化模式不断在一个又一个的行业重现。只要存在对某种商品的需求，技术就会发挥作用，将这种商品的价格变得很低，从而可以大

① 一块普通的石英表，每天的误差大约是0.5秒。走时最准的百达斐丽牌机械表，每天的误差是−1.5秒到+2秒，零售价是22万美元。

② 产于1989年、保存状况良好的林肯牌加长型礼车，现价大约是5000美元。产于2004年的奔驰S600轿车的价格是12.2万美元。如果要一个生活在20世纪初的普通人分辨哪一辆车价格更贵，他大概会猜错。

量销售。[1]一旦产品能够流水线生产，即使质量没有改进，至少也会更便于使用。富人最喜欢的就是那些方便易用的产品。我认识的富人朋友，与其他朋友相比，开着同样的车，穿着同样的衣服，使用同样的家具，吃着同样的食品。虽然他们的房子是在不一样的地方，或者即使与普通人在同一个社区，面积也要大得多，但是他们的生活确实与普通人是一样的。房子的建造方法也是一样，屋里的东西也基本接近。拥有定制的昂贵商品反而不方便。

富人日常做的事情也和普通人差不多。无所事事的闲适生活早就成为罕见情况了。如今，确实有很多人非常有钱，完全不必再去工作，他们之所以还在工作，不是因为感到社会压力，而是因为无所事事使人感到孤独和消沉。



今天的社会身份（social distinction）差异也要比 100 年前来得小。那时的小说和讲解礼仪的手册在今天读起来好像是在说陌生的部落社会。Beeton 夫人出版于 1880 年的《家务手册》（*Book of Household Management*）这样写道：“至于说到朋友之间的友谊……在某些情况下，为了承担家庭生活的责任，女主人可能必须放弃一些她早年认识的朋友。”一个女人嫁给了有钱人，就被认为应该放弃那些没钱的朋友。要是你今天这样做的话，别人会觉得你的行为很野蛮，而且你也会让自己过上一种乏味无趣的生活。今天的人们多多少少还是有一些互相隔离的趋

---

① 如果想要真正地对收入加以考察，你必须使用“真实收入”的概念（以购买力衡量的收入），而不是使用“名义收入”的概念（以货币衡量的收入）。但是，计算“真实收入”的常用方法忽略了大部分随着时间增长的财富，因为“真实收入”要用消费者物价指数才能算出来，但是消费者物价指数是根据一系列样本商品的价格计算的，本身就不具有全面的精确性，而且新发明产品的价格没有计算在内。（只有当新发明产品成为价格稳定的常用物品后，才会计算在内。）

所以，就算我们认定有了抗生素、飞机旅行、电力系统以后，人类的生活大大改善，真实收入的计算方法却说我们的生活只有轻微的改善。

衡量收入变化的另一种方法就是，问如果你乘坐时间机器回到过去，你需要花多少钱购买同样的东西。举例来说，假定你回到1970年，你会发现今天价格不到500美元的CPU处理能力在那时至少价值1.5亿美元。这种价格的衰变随着时间流逝很快就会接近于零，因为一百年后，你今天需要的所有东西后人都不会想要。相反，如果你把今天可乐饮料的宝特瓶拿回到1800年，它会被认为是精美的工艺品。

势，但主要是因为教育层次的差别，而不是财富的差别。[①]

无论在物质上，还是在社会地位上，技术好像都缩小了富人与穷人之间的差距，而不是让这种差距扩大了。如果参观雅虎、英特尔、思科的办公室，会看到每个人都穿着差不多的衣服，有着同样的办公室（或者小隔间）、同样的家具，彼此直呼对方的名字，不加任何头衔或敬语。表面看大家没什么差距，但如果看到每个人银行户头上的余额差别如此之大，一定会感到震惊不已。

技术的发展加大了贫富差距，这是不是一个社会问题？好像没有那么严重。技术在加大收入差距的同时，缩小了大部分的其他差距。

公理的不同意见

你经常可以听到有人批评某种政策会加剧贫富分化。隐藏的意思就是，贫富分化的加剧一定是坏事，这好像已经成了公理。收入差距的扩大可能确实不好，可是我不觉得这可以被看成公理。



实际情况是，在工业化的民主国家，这种观点更可能是错误的。在农奴和贵族组成的社会，收入差距的加大肯定是社会问题加剧的信号，收入更多地从农奴流向了贵族。但是，抢夺他人的财富已经不再是收入的唯一来源了。波音 747 飞机驾驶员的收入大概是商场收银员的 40 倍，但是前者不是贵族，后者也不是奴隶，这种收入差距只是因为前者的技能比后者的要值钱得多。

我想提出一种相反的观点：现代社会的收入差距扩大是一种健康的信号。技术使得生产率的差异加速扩大，如果这种扩大没有反映在收入上面，只有三种可能的解释：（a）技术革新停顿了；（b）那些创造大部

① 有人会说，教育程度的差别与财富的差别是一回事，因为富人得到高等教育的机会更大。这个论点是成立的，某种程度上可以做到用钱把孩子送进顶尖的大学。你只要把孩子送进昂贵的私立学校，就等于敲开了大学的门。

根据2002年美国国家教育统计中心的一份报告，大约1.7%的美国儿童就读于私立的非教会学校，而普林斯顿大学2007级新生中，大约有36%的人来自于这样的学校。（有趣的是，哈佛大学的这个比例要低不少，只有大约28%。）很显然，这是教育制度的一个大漏洞，但是它正在缩小，而不是扩大。

也许，大学入学申请制度的设计者应该参考计算机安全行业的做法，不要假设你的系统毫无漏洞，而是搞清楚多大程度上漏洞无法被利用。

分财富的人停止工作了；（c）创造财富的人没有获得报酬。

我觉得可以很有把握地说，（a）和（b）都不是好事。如果你有不同意见，那不妨试试去过公元九世纪法兰克王国的贵族生活，一年后再来告诉我们你的感受。（我很仁慈地没有建议你去过石器时代的那种生活。）

如果你想让社会保持繁荣，同时收入差距不扩大，那么就只剩下（c）这一种可能了，即创造大量财富的人不获取报酬。举例来说，苹果公司的两位创始人将欢欣鼓舞地每天工作 20 个小时，为社会提供苹果电脑，然后只领取一份相当于大公司里朝九晚五的上班族领取的税后工资。

如果得不到报酬，人们是否愿意创造财富？唯一的可能就是，工作必须能提供乐趣。会有人愿意免费写一个操作系统，但是他们不愿意免费为你安装、提供电话支持、进行客户培训等。即使是最先进的高科技公司，也有至少 90%的工作没有乐趣、令人生厌。



在一个剥夺个人财产的社会，财富创造活动中所有那些没有乐趣的事情都会急剧地放慢，乃至停顿。对历史进行实证检验，我们就可以得出这个结论。假设你听到一种噪音，你觉得是身边的电扇发出的。你关了电扇，噪音停止；打开电扇，噪音又出现。关了就安静，打开就嘈杂，就是这样一种情况。如果没有其他信息，看上去噪音就是由电扇发出的。

在历史的不同时点，你是否能够通过创造财富而致富就是这样一种打开/关闭的循环过程。公元 800 年的意大利北部，关闭（贵族偷窃平民的财产）；公元 1100 年的意大利北部，打开；公元 1100 年的法国中部，关闭（仍然是封建社会）；公元 1800 年的英国，打开；1974 年的英国，关闭（投资所得税高达 98%）；1974 年的美国，打开。以上每一种情况，打开时，财富创造活动就出现了，关闭时，财富创造活动就消失了，这就好像电扇和噪音的那种相伴关系。

社会的变化涉及很多因素，并不仅仅是由于创造财富的原因。有很多因素发挥作用。如果研究对象只是一台电扇，那么不用考虑太多别的因素就能断定噪音是由电扇发出的，但是研究财富问题就没有这么简单了，必须要考虑很多别的因素。

但是，只要你压制收入差距的扩大，不管是用偷窃私人财产的做法

（封建社会），还是用高额税收的做法（某些现代政府），最终结果看来都是一样的，那就是社会作为一个整体变得更贫穷了。

如果我可以做选择，到底是生活在一个整体上非常富裕但是我个人相对贫穷的社会，还是生活在一个我个人相对非常富裕但是整体上非常贫穷的社会呢？我会选择第一个选项。如果我有小孩的话，可能哪一个选项更好还值得争论。但是，总的来说，你要避免的是绝对贫穷，而不是相对贫穷。如果必须在这两种社会之间做选择，根据目前的证据，我选择个人相对贫穷、但是整体上更富裕的社会。

一个社会需要有富人，这主要不是因为你需要富人的支出创造就业机会，而是因为他们在致富过程做出的事情。我在这里谈的不是财富从富人流向穷人的那种扩散效应（trickle-down effect），也不是说如果你让亨利·福特致富，他就会在下一场宴会雇用你当服务员，而是说如果你让他致富，他就会造出一台拖拉机，使你不再需要使用马匹耕田了。



8

防止垃圾邮件的一种方法

我认为过滤垃圾邮件是可以做到的，基于内容的过滤器将发挥作用。发送垃圾邮件的人有一个致命伤，那就是他们发送的邮件本身。他们有办法逃脱你搭建的其他壁垒（至少目前是这样），但是不管怎样，他们都必须把垃圾邮件发出去。如果我们能够写出可以从内容上识别出垃圾邮件的软件，那么他们就无法逃脱了。[①]

收信人很容易识别哪些是垃圾邮件，哪些是正常邮件。如果你雇人用肉眼帮你清除垃圾邮件，这事情应该没有太大难度。那么我们怎么用软件自动模拟这个过程（假定不使用复杂的人工智能）？

我觉得只用一些很简单的算法就可以做到这一点。事实上，我发现只要对单个词语进行贝叶斯判断，就能很好地过滤大部分垃圾邮件。设置好贝叶斯过滤器（详见后文），1000 封垃圾邮件能够被过滤掉 995 封，并且没有一个误判。

开发垃圾邮件过滤器时，统计学方法往往不是程序员首先想到的方法。大多数黑客的直觉是写出一个能够识别垃圾邮件某种特征的软件。你看着那些垃圾邮件，心想这些可恶至极的家伙胆敢向我发送以“亲爱的朋友”开头的邮件，或者主题行都是大写字母且以八个惊叹号作为结尾的邮件，我用一行代码就能把它们全过滤掉。

你这样做了以后，一开始效果还不错。几条简单的规则就能拦截大

① 出版时，本文的一些内容经过改写，但是从Lisp代码翻译过来的、计算垃圾邮件概率的数学公式没有变。所以，公式里有些地方可能过时了，比如现在很少有垃圾邮件还含有click这个词。但是，算法仍然是有效的。一个略加修改的版本可以过滤99.6%的垃圾邮件，更多信息参见paulgraham.com。

部分垃圾邮件。仅仅搜索单词 click 就会捕捉到 79.7%的垃圾邮件（以我的情况为例），其中只有 1.2%是误判。

在转向统计学方法之前，大约整整有六个月，我一直使用这种特征过滤法，自己编写软件，识别垃圾邮件的特征。我发现，到后来要想把识别精度提高几个百分点非常困难，如果我把过滤条件设置得很严格，误判率就会上升。

所谓误判，指的是正常的邮件被错误认定为垃圾邮件。对于大多数用户来说，错过一封正常的邮件后果要比收到垃圾邮件严重得多。所以，如果过滤器有误判，就好像治疗粉刺的药物却有致人死亡的危险一样。

用户收到的垃圾邮件越多，他就越不可能注意到被过滤掉的垃圾邮件中包含着一封正常邮件。这就导致了一个很奇怪的后果，如果你的过滤器效果越好，就越不能出现误判，一旦误判，后果就会变得很严重，因为过滤器工作得非常良好，所以用户相信它，就不太可能去检查被它过滤掉的邮件。



我不知道为什么我没有早一点尝试统计学方法。原因可能是我太过迷恋于发现垃圾邮件的特征，有一种与发送者斗智斗勇的感觉。（大多数黑客都是好胜心很强的人，一般人往往意识不到这点。）当我尝试统计学方法以后，我立刻发现这是更聪明的选择。它不仅能发现普通的垃圾邮件标志（比如，木马和广告性词语），还能发现像per、FL、ff0000 这种不太明显的标志。事实上，ff0000（HTML语言中表示鲜红色的代码）被证明效果显著，能很有效地识别垃圾邮件，就像色情词汇一样容易辨别。

下面我就简单介绍一下我是如何开发统计学过滤器的。开始前，我先准备好一组垃圾邮件和一组非垃圾邮件，每组各有 4000 个样本。我对每一封邮件的全部内容进行了扫描，包括邮件头、内嵌的 HTML 代码和 JavaScript 代码。我把字母、阿拉伯数字、破折号、撇号、美元符号作为“实义标识”（token），所有其他字符则是“实义标识”的分隔符。（这个处理可能还可以进一步改善。）我忽略了完全由数字组成的字符串以及 HTML 注释，也不把它们当作“实义标识”的分隔符看待。

我计算了每个实义标识在两个邮件组出现的次数（忽略大小写）。完成这步以后，我就得到了两大张散列表，一个邮件组一张，表中每一栏就是一个键值对，“键”栏对应每一个实义标识，“值”栏则是这个标识出现的次数。

接着，我创建了第三张散列表，“键”栏还是每一个实义标识，“值”栏则是包含该标识的邮件是垃圾邮件的概率。我把这个概率记作 $P_{spam|w}$，计算公式如下：

$$r_g = \min(1, 2(good(w) / G)), \quad r_b = \min(1, bad(w) / B)$$

$$P_{spam|w} = \max(0.01, \min(0.99, r_b / (r_g + r_b)))$$

公式中的 w 就是我用来计算概率的那个实义标识，*good* 和 *bad* 表示我在第一步创建的两张散列表，G 和 B 分别表示正常邮件和垃圾邮件的数量。



为了避免误判，我稍微加大了某个实义标识不是垃圾邮件的概率。经过反复试错，我发现将 *good* 表的次数值全部增大一倍可以很好地达到这个目的。这有助于区分那些偶尔出现在正常邮件中的词以及那些几乎从不出现的词。我只把出现总次数超过 5 次的词列入计算范围（实际上，由于正常邮件会反复使用同样的词，所以出现总次数超过 3 次应该就够了）。下一个问题就是，如果一个词只出现在一组邮件中，它的概率应该怎么分配。我又通过试错法选择了 0.01 和 0.99。这里可能还有改善的余地，但是随着邮件数量的增加，计算结果应该会自动调整的。

那些善于观察的人会注意到为了计算每个词出现的次数，我把每一组邮件看成一整串文本流，但却还是使用电子邮件的数量而不是文本流的总长度作为计算概率时的分母。这样做也是为了加大不是垃圾邮件的概率，防止出现误判。

当收到新邮件的时候，程序会自动扫描，读出邮件中所有的实义标识，再找出其中 15 个最醒目标识（所谓“最醒目标识”，就是指概率偏离中性值 0.5 最远的标识），用它们判断整封邮件是垃圾邮件的概率。如果用 $w_1, \cdots, w_{15}$ 分别表示 15 个最醒目标识，那么计算整封邮件概率的公式如下：

$$P_{spam} = \frac{\prod_{i=1}^{15} P_{spam|w_i}}{\prod_{i=1}^{15} P_{spam|w_i} + \prod_{i=1}^{15} (1 - P_{spam|w_i})}$$

实践中遇到的问题是，如果出现一个以前从来没见过的词（即两张散列表里都找不到这个词），它的概率应该怎么计算。我发现（还是通过试错法）将概率设为 0.4 效果很好。如果你从来没见过这个词，它多半是一个正常的词，垃圾邮件用的词都是很常见的。

如果上面的公式计算出来的概率大于 0.9，我就把这封邮件当作垃圾邮件。但是在实践中，把这个门槛值设为多少并不是很重要，因为计算出来的概率值大多数都分布在两端，很少落在中间。

统计学方法的一大优点就是，你不需要一封封去看垃圾邮件。在使用它之前的六个月，我大概看了足足几千封垃圾邮件，这真是很苦恼的一件事。数学家 Norbert Wiener 说，如果你与奴隶比赛，你也会变成一个奴隶。与垃圾邮件搏斗就有这种令人退化的效果。为了识别垃圾邮件的每一个特征，你不得不钻进发送者的脑袋，搞清楚他们怎么想。说实话，我一刻都不想待在那里。

但是，贝叶斯方法的真正优点在于你知道你正在计算的是什么东西。识别垃圾邮件特征的过滤器（比如SpamAssassin）为每封邮件计算一个“得分”，而贝叶斯方法为每封邮件算出一个概率。“得分”方法的缺点在于没人知道这个分数到底是什么意思，用户不知道，更糟的是，就连过滤器的开发者也不知道。如果邮件中有sex（性）这个词，请问得分是多少？计算概率当然也会出错，但是至少意义上很清楚，一点也不模糊，而且用来计算它的那些依据也很清楚。根据我的邮件库，一封邮件中含有sex这个词，那么它有 0.97 的概率是一封垃圾邮件；要是含有sexy这个词，垃圾邮件的概率更是上升到 0.99。贝叶斯规则同样毫不含糊地表明，如果一封邮件同时含有这两个词，即使没有其他证据（事实上，这是不可能的），垃圾邮件的概率也将达到 99.97%。

因为贝叶斯方法计算的是概率，所以它必须考虑邮件中所有的线索，不管是肯定性线索还是否定性线索。有些词（比如though、tonight、



apparently）极少出现在垃圾邮件中，所以它们会大大降低这封邮件属于垃圾邮件的概率；同样，还有一些词（比如unsubscribe、opt-in）几乎是垃圾邮件专用，它们会大大增加概率。因此，如果一封邮件的其他方面都合格，只是碰巧包含了sex这个词，这封邮件是不会被归入垃圾邮件的。

理想情况下，每个收信人应该都有自己单独的概率分布表。以我为例，我收到的许多邮件中都含有 Lisp 这个词，而迄今还没有垃圾邮件包含这个词。所以，一个这样的词实际上就像许可证一样，保证了这封信是发送给我的正常邮件。在我以前写的垃圾邮件过滤器中，用户可以自己开出一张清单，列出一系列这样的词。然后，收到的邮件之中如果包含这些词，就将自动通过过滤器。我自己的清单上除了 Lisp 这个词，还有我的邮政编码，所以网上购物的确认邮件就能安然通过过滤器（否则它们看上去很像垃圾邮件）。我当时觉得自己真是聪明绝顶，但是后来发现贝叶斯方法能够自动做到这一点，而且它还能发现许多我以前根本没意识到的这一类词语。



我在文章的开头说，我的过滤器现在可以在 1000 封垃圾邮件中正确识别出 995 封，并且没有一个误判。做到这一点的前提是必须有一个很大的邮件库作为判断依据。但是，我不想用这些数字误导读者，如果你想同样做到这个水平，最好采用我提倡的方法，就是把自己收到的所有邮件分成垃圾邮件和非垃圾邮件两大类。按照我的想法，每个用户应该有两个“删除”按钮，一个是“正常删除”，还有一个是“垃圾邮件删除”。任何被后一个按钮删除的邮件都进入垃圾邮件库，而其他的所有邮件进入非垃圾邮件库。

刚开始的时候可以有一个所有人共享的基本概率分布表，但是到了最后，每个用户应该都分别有自己的概率分布表，这是根据他收到的邮件对每一个词进行统计后得出的。这样做可以：(a) 使得过滤器更有效；(b) 让每个用户自己定义，什么是他眼中的垃圾邮件；(c) 使得垃圾邮件的发送者无法针对过滤器做出调整（这可能是最大的好处）。如果每个用户的过滤器大部分都是基于独立的数据库，那么每个过滤器的过滤条件都不一样，而且会更加富有成效。要是垃圾邮件的发送者仅仅针对基

本概率分布表做出调整，并不能保证这封邮件会通过拦截。

统计学过滤器除了基于内容做出判断以外，还可以有一张白名单，上面列出值得信任的、不会发送垃圾邮件的发信人，让他们的邮件直接通过过滤器。建立这样一张白名单有一个容易的方法，就是将所有你曾经去信的地址都保留下来。另外，凡是你使用“正常删除”按钮删除的邮件（前提是邮箱软件必须同时具备“垃圾邮件删除”按钮），它们的地址也可以加入白名单。

我提倡使用白名单，主要是为了节约计算，而不是认为这样可以改进过滤器的效果。我曾经认为白名单会让过滤器运作得更顺利，因为你从此只需要扫描那些陌生人的邮件就行了。试想一下，如果某人是第一次发邮件给你，他一般囿于常规，只会说一些需要对你说的内容，不会一上来就跟你讨论sex。相反，倒是你已经认识的熟人可能会这样做。所以，白名单有助于避免这些邮件的误判。但是问题是，人们一般都有好几个Email地址，一封从陌生地址发来的邮件并不必然意味着来自一个你不认识的陌生人。一个老朋友突然用一个全新的地址写信给你可不是罕见情况，对于黑客尤其如此。所以，白名单并不会降低误判的风险。



不过，某种意义上，统计学过滤器其实内嵌了白名单（还有黑名单）。因为整封邮件都会被扫描，包括邮件头在内，所以经过这一步，过滤器自己“知道”哪些邮箱地址可以信赖（甚至还知道哪些中转的服务器可以信赖）。对于垃圾邮件，它也会“知道”得一清二楚，包括服务器名称、发送邮件的软件版本和邮件协议。

如果现在的过滤水平（1000封垃圾邮件识别出995封）可以保持下去，我会觉得问题已经解决了。但是，垃圾邮件永远在进化，现在能够过滤它们不等于永远能够过滤它们。说实话，如今的大多数垃圾邮件过滤器就像杀虫剂一样，唯一作用就是创造出杀不死的新品种害虫。

我对贝叶斯方法寄予厚望，因为它的过滤能力可以随着垃圾邮件一起进化。所以，假定垃圾邮件发送者开始用v1agra替代viagra[①]，以此逃避某些机械的、基于单个词汇的过滤器的拦截，贝叶斯过滤器却能够自

① 中文名“万艾可”（伟哥），一种治疗阳痿的药物。——译者注

动注意到这种变化。实际上，v1agra 是比 viagra 确定性高得多的线索，可以证实这封邮件为垃圾邮件，至于概率到底高出多少，贝叶斯过滤器将准确告诉我们。

到目前为止还存在一个问题，所有垃圾邮件过滤器的开发者必须回答：如果发送人准确知道你的过滤机制，他们逃避拦截的可能性有多大？比如我猜想，如果“校验码”（checksum）方法[①]对垃圾邮件构成重大威胁，那么发送人就会耍花招，使用同义词替代的技巧让每一封邮件内容完全不同，从而逃避拦截。

但是，要想骗过贝叶斯过滤器就没那么容易了。你把每一封垃圾邮件都写得独一无二或者不使用某些特定的标志性词汇，都不足以达到目的。只有让垃圾邮件看上去与正常邮件毫无区别才能够实现。我觉得要做到这一点真是够难为他们的。垃圾邮件主要用于销售目的，那么除非你正常往来的邮件都是销售类邮件，否则垃圾邮件不可避免地将与其他邮件不一样。此外，发送人还必须改变（并且不断改变）他的邮件系统架构，否则贝叶斯过滤器会识别出他的邮件头，而根本不用看邮件内容到底写的是什么。我对邮件系统架构知道得不多，不太清楚让邮件头逃过拦截的难度有多高，但是我猜想它的难度要超过让邮件正文逃过拦截的难度。



假定那些人连邮件头的难题也解决了，那么未来的垃圾邮件可能就是下面这个样子：

```
嗨，你好。请查看链接：
www.27meg.com/foo
```

这差不多就是统计学过滤器能够允许通过的销售类邮件的样子，最多就到这样了。（可是实际上，这段话更难逃过拦截，因为邮件的其他内容全部都是中性词语，垃圾邮件可能不得不在 URL 上做文章，但是要让一个 URL 看上去没有可疑之处还是很伤脑筋的。）

发送垃圾邮件的人形形色色。有的是公司，经营着一个所谓的邮件

① “校验码”方法的原理是，一般来说，垃圾邮件都是大量群发的，除了个别词语不同以外，信件的主体内容完全一样。所以，只要去除那些不同的部分，对信件主体计算一个校验码，然后与数据库中已经确认的垃圾邮件校验码进行比较，如果两者相同，就可以认定是垃圾邮件了。——译者注

列表，表面上说你可以选择订阅，但是实际上根本无法退订，他们肆无忌惮地向你发送广告；有的是个人，专门劫持邮件服务器，推广色情网站。如果我们的过滤器迫使他们只能把垃圾邮件写成上面那样，应该会使得垃圾邮件业中合法经营的那部分人退出这个行业。因为他们很乐于遵守各州的法律规定，在邮件中附上正式声明，解释为什么自己不是垃圾邮件以及如何才能取消订阅。这一类文字反而使得识别他们变得更容易了。

（我以前曾经认为，那些相信更严格的法律会遏制垃圾邮件的人真是太天真了。我现在认为，更严格的法律或许无法减少我们收到的垃圾邮件的数量，但是肯定有助于减少逃过过滤器拦截的垃圾邮件的数量。）

在垃圾邮件业中，如果发送销售类垃圾邮件受到限制，那么整个行业将不可避免地受到重创。"行业"这个词是很准确的，发送垃圾邮件的人其实都是商人，他们这么做只是因为这招很有效。虽然垃圾邮件的回应率低到不能再低了（不超过百万分之 15，相比之下，传统的邮寄商品目录的回应率是百万分之 3000），但是发送垃圾邮件的成本实际上为零，所以它还是有效的。但是对于收到垃圾邮件的人来说，成本却很高昂，假定有 100 万人分别收到一封垃圾邮件，每人花一秒钟删除，累计起来就相当于一个人 5 个星期的工作量，而发送人连一分钱也不用付出。

不过，虽然接近于零，发送垃圾邮件还是有成本的。[①]所以，只要我们把垃圾邮件的回应率降得很低（不管手段是直接过滤，还是让垃圾邮件被迫掩盖它们的销售意图），商家就会发现，发送垃圾邮件是一件经济上不值得的事情。

另一方面，垃圾邮件使用了那么多推销语言就是为了增加回应率。如果有一天推销语言突然不能用了，对他们就是重大打击。为了说明这一点，让我们把自己想象成一个回应垃圾邮件的人，看看这些人到底是怎么想的（这要比把自己想象成垃圾邮件发送者更让人难受）。回应垃圾



① 2002年，发送100万封垃圾邮件的最低成本好像是200美元。这个价格很便宜，相当于每封垃圾邮件成本为0.02美分。但是，假定过滤器可以拦截95%的垃圾邮件，那么要使得受众数量保持不变，发送人的成本就必须增加20倍。而垃圾邮件推销的那些生意，利润几乎肯定到不了这么高，无法抵消这笔成本。

邮件的人要么是惊人地轻信，要么是表面上完全否认、但是私底下却有着对性的强烈兴趣。不管哪一种情况，也不管垃圾邮件在正常人看来是多么令人反感或愚蠢万分，总是可以让这些人兴奋不已，因为邮件内容写得实在太诱人了，毕竟如果不是这样，商家也就不发送垃圾邮件了。要是邮件内容改成“请点击下面的链接”，对于收信人来说就没有太大的吸引力了，根本比不上现在的效果。结果就是，如果垃圾邮件不能使用诱人的推销语言，它作为推销工具的价值就会大大降低，使用它的商家数量也会减少。

最终，我们将取得全胜。我开始写垃圾邮件过滤器只是因为不想再让这些东西烦我了。但是，如果我们把过滤器做得足够好，那么垃圾邮件将不再有效，商家最后将不再发送它。



在所有对抗垃圾邮件的方法之中（从软件方法到法律方法），我认为单独来看，“贝叶斯过滤”是最有效的工具。但是，我也认为，我们使用的不同方法越多，综合效果就越好，因为任何对发送人构成限制的方法往往都会使得过滤器工作起来更顺利。即使同样是基于内容的过滤器，我也认为，如果有多种不同的软件可以同时使用会比较好。过滤器的差异越大，垃圾邮件想要逃过拦截就越不可能。

9

设计者的品味

哥白尼不认同托勒密的体系，一个极其重要的原因是，他觉得托勒密提出的偏心等距点（equant）毫无美感……

——托马斯·库恩，《哥白尼革命》

我们所有人都受到凯利·约翰逊[①]的影响，狂热地相信外观优美的飞机一定会飞得同样漂亮。

——本·里奇，《臭鼬计划》

美感是第一道关卡。丑陋的数学在世界上无法生存。

——G. H. 哈代，《一个数学家的辩白》



最近，我与一个在MIT教书的朋友交谈。他的研究领域很热门，每年申请他的研究生的人多得让他应付不过来。“很多人看上去很聪明，”他说，“但是我不知道他们的品味如何。”

品味。如今很少听到这个词了，人们往往使用别的叫法，但它却的的确确是我们离不开的基本概念。我的朋友的意思是，他想要的学生不仅应该技术过硬，还应当能够使用技术做出优美的产品。

数学家会把出色的工作称赞为“优美的”。无论古今，科学家、工程师、音乐家、建筑师、设计师、作家、画家都是这样做的，他们都使用同一个词。这仅仅是巧合吗，还是他们之间有共识？如果真的有共识，那么我们能不能将某一个领域发现的“美”的规律运用于另一个领域呢？

① Kelly Johnson（1910—1990），美国传奇飞机设计师，供职于洛克希德公司，主导设计的机型达40余种。——译者注

对于我们设计师来说，美就不仅仅是一个理论问题了。如果世界上真有“美”存在，我们需要能够认出它。设计产品时，我们需要良好的品味。与其把“美”说成一个虚无缥缈的抽象概念，还不如让我们考虑一个实际的问题（这样就能避免喋喋不休的空谈）：如何才能做出优美的产品？

如果你在当今社会提到“品味”，很多人会对你说“品味是主观的”。他们真的就是这么认为的。喜欢一件东西，却不知道为什么自己喜欢它，原因可能是这件东西是美的，但也可能因为他们的母亲也拥有同样的东西，或者杂志上某个明星使用它，或者仅仅因为它的价格很昂贵。人类的思想就是没有经过整理的无数杂念的混合。

我们大多数人从孩提时代起就被鼓励不要去分析清楚自己的头脑。如果你的小弟弟画图时把人都涂成绿色，你想取笑他，你妈妈很可能会对你说：“你有你喜欢的方式，他有他喜欢的方式。”



你妈妈这时不是教给你什么是美学，而只是想阻止你们两个争吵。

就像大人哄小孩的其他话一样，这句话也是模棱两可的，与其他话会发生冲突。大人教导你说品味只是每个人的偏好而已。但是来到博物馆，他们却对你说，仔细观赏达·芬奇的作品，因为他是伟大的艺术家，品味超凡。

小孩子受到这样的教导会怎么想？他会怎么理解“伟大的艺术家”？这么多年来，别人无数遍地告诉他，品味就是一种偏好，是每个人自己的事情，所以他不可能直接就明白，所谓“伟大的艺术家”就是这个人的作品要比其他人的杰出。他更可能觉得，所谓“伟大的艺术家”只是针对我个人世界而言的，就是很符合我自己口味的艺术家，好比某本书上说食用西兰花对我的健康有利，所以我就应该喜欢吃西兰花一样。

把品味说成个人的偏好可以有效地杜绝争论，防止人们争执哪一种品味更好。但是问题是，这种说法是不正确的。只要你自己开始动手设计东西，就能明白这一点。

不管每个人的工作是什么，他们内心里都有一种愿望——把自己的

工作做好。足球运动员想赢得比赛，CEO 想增加利润。做好自己的工作会真正令人感到自豪和愉快。但是，如果你是一个设计师，并且你不承认有一种人们共同认可的东西叫做“美”，那么你就没有办法做好工作。如果品味只是一种个人偏好，那么每个人都是完美无缺的：你喜欢自己看上的东西，那就足够了。

就像别的工作一样，只要你不断地从事设计工作，你就会做得越来越好。你的品味会出现变化，你会像别人一样有所提高。如果这样的话，那么你以前的品味就不只是与现在不同，而是不如现在的好。因此，所谓的“品味没有好坏之分”的公理也就顿时见鬼去了。

现在流行“相对主义”，即认为真理是相对的。即使你已经从小孩变成了成年人，这种观点依然可能妨碍你思考“品味”。但是，只要你走出狭隘的自我，至少在心里对自己说，确实存在比其他设计更好的杰出设计，那么你就能开始仔细研究了。你的品味是如何变化的？什么原因使你做出不好的设计？其他人对设计是什么观点？

只要你开始思考这些问题，你就会发现，众多不同学科对“美”的认识有着惊人的相似度。优秀设计的原则是许多学科的共同原则，一再反复地出现。



好设计是简单的设计。从数学领域到绘画领域，你都可以听到这种说法。在数学中，它表示简短的证明往往是更好的证明。特别是对于数学公理来说，少即是多。在编程中，这种说法也基本适用。对于建筑师和设计者，它意味着美依赖于一些精心选择的结构性元素，而不依赖于表面装饰品的堆砌。（装饰品本身并不是坏事，只有当它被用来掩盖结构的苍白时，才变成了一件坏事。）绘画也是类似的，认真观察的、非常有代表性的静物作品往往要比表面极尽华美、但是实质上只是无意义重复的“巨作”（比如再现非常复杂的花边的绘画作品）更有价值。在写作上，这种说法意味着只说必须要说的话，并且说得简短。

这样强调简单似乎有点奇怪。有人会说，简单就是事物本来的样子，装饰反而意味着更多的工作。但是，当人们自己从事创造性工作的时候，好像就会忘了保持简单这个原则。刚开始写作的人喜欢用浮夸的语调，

根本不像他们平时说话的样子。设计师喜欢用波浪式卷曲表现他们的艺术感。画家发现自己都是表现主义者（expressionist）。这些装饰都是花架子，在作家的长句、画家“表现主义”的画笔之下，根本就是空洞无物，表面的装饰掩盖了内部的空虚，太可怕了。

当你被迫把东西做得很简单时，你就被迫直接面对真正的问题。当你不能用表面的装饰交差时，你就不得不做好真正的本质部分。

好设计是永不过时的设计。只要没有错误，每一个数学证明都是永不过时的。所以，数学家哈代才会说：“丑陋的数学在世界上无法生存。”他的意思与飞机设计师凯利·约翰逊的观点是一样的：如果解决方法是丑陋的，那就肯定还有更好的解决方法，只是还没有发现而已。

以永不过时作为目标是一种帮助自己找到最佳答案的方法：如果你不愿别人的答案取代你的答案，你就只好自己做出最佳答案。某些大师的作品太过杰出，永不过时，使得后人几乎难以在该领域立足。自从 16 世纪出现了德国雕刻大师丢勒（Dürer），后世的雕刻家都因为自己的作品被拿来与他的作品作比较而苦不堪言。



以永不过时作为目标也是一种避开时代风潮的影响的方法。“风潮”这个词，从字面上就可以看出，它就是一阵风似的，随着时间经常改变。如果一件东西长盛不衰，那么它的吸引力一定来自本身的魅力，而不是来自风潮的影响。

说来奇怪，如果你希望自己的作品对未来的人们有吸引力，方法之一就是让你的作品对上几代人有吸引力。我们很难猜想未来是什么样子，但是可以肯定，未来的人们不会在乎今天流行的风潮，这一点与上几代人是相同的。所以，如果你的作品对今天的人们以及 1500 年的人都有吸引力，那么它极有可能也会吸引 2500 年的人。

好设计是解决主要问题的设计。厨房的煤气灶有四个出火口，排成一个正方形。每个出火口都由一个调节器控制，四个出火口就有四个调节器。请问应该如何摆放调节器？最简单的摆放方法当然是把四个调节器排成一列，但要是这样做，人们使用起来就很不方便，每次都要停下

来想一下到底每个调节器对应的是哪个出火口。如果直接把调节器排成与出火口一样的正方形，就不会有这个问题了。

许多坏设计做得很辛苦，但是从一开始方向就错了。20 世纪中期，有一股使用无衬线（sans-serif）字体的潮流。这一类字体接近于纯手写的样式，但是它无助于解决最主要的问题。印刷出来的文字首先应该是易于辨认的，所以能够清晰地分辨字母就是最主要的问题。传统的新罗马（Times Roman）字体是一种有衬线的字体，虽然看上去古老得就像维多利亚女王时代的风格，但是它的小写 g 就是可以很轻易地与小写 y 区分。

答案可以不断改进，同样，问题本身也可以不断改进。软件的难题通常可以被改成等价的较易解决的形式。历史上，物理学的主要难题曾经一度是如何诠释经典著作,后来逐渐变成对可观测到的行为进行预测，这种转变使得物理学的发展速度大大加快。



好设计是启发性的设计。英国女作家简・奥斯汀的作品几乎不带有任何描述。她不告诉读者每件东西看上去是什么样子，只是把故事讲得非常生动，让读者自己把一切都想象出来。同样，绘画作品也分为描述性绘画和启发性绘画，后者往往比前者更引人入胜。每个人看到《蒙娜丽莎》都有自己的理解。

在建筑学和设计学中，这条原则意味着，一幢建筑或一个物品应该允许你按照自己的愿望来使用。举例来说，一幢好的建筑物应该可以充当平台，让你想怎么布置就可以怎么布置，过上自己想过的家庭生活，而不是使得你像执行程序一样只能过上建筑师为你安排的生活。

在软件业中，这条原则意味着，你应该为用户提供一些基本模块，使得他们可以随心所欲自由组合，就像玩乐高积木那样。在数学中，这条原则意味着，一个可以成为许多新工作基础的证明要优于一个难度很高、但无助于未来学科发展的证明。在科学领域中，总体上可以把引用次数看作对他人启发性大小的粗略指标。

好设计通常是有点趣味性的设计。这条原则可能不是所有情况下都

成立。但是，丢勒的雕刻、芬兰设计师沙里宁（Saarinen）的子宫椅（Womb Chair）、意大利罗马的万神殿（Pantheon）、保时捷 911 型汽车的原型设计（图 9-1），在我看来都很有趣。逻辑学家哥德尔（Gödel）的不完备定理就好像一个玩笑那样有意思。



图 9-1　保时捷 911E，1973 年产



我想，这是因为幽默一定程度上反映了力量。幽默感是强壮的一种表现，始终拥有幽默感就代表你对厄运一笑了之，而丧失幽默感则表示你被厄运深深伤到。所以，强壮的标志（或者至少是特点）就是轻松面对自己的人生。充满自信的人常常像燕子一样，以一种居高临下的姿态轻盈地看待周围的一切，比如希区柯克拍摄的电影、16 世纪画家勃鲁盖尔（Bruegel）的绘画（甚至莎士比亚也是一个这方面的例子）。

好的设计并非一定要有趣，但是很难想象完全无趣的设计会是好的设计。

好设计是艰苦的设计。如果观察那些做出伟大作品的人，你会发现他们的共同点就是工作得非常艰苦。如果你工作得不艰苦，你可能正在浪费时间。

困难的问题需要艰巨的付出才能解决，高难度的数学证明需要结构非常精细的解决方法（它们往往做起来很有趣），工程学也是如此。

当你攀登高山时，必须扔掉一切不必要的装备。在困难地点或预算不足的条件下，建筑师就只能做出很简练的设计。当解决难题成为压倒一切的任务时，那些流行样式与华丽装饰就被抛到一边去了。

并非所有的痛苦都是有益的。世界上有有益的痛苦，也有无益的痛苦。你需要的是咬牙向前冲刺的痛苦，而不是脚被钉子扎破的痛苦。解决难题的痛苦对设计师有好处，但是对付挑剔的客户的痛苦或者对付质量低劣的建材的痛苦就是另外一回事了。

在绘画上，肖像画通常占据最高地位。这不是偶然的，原因不仅是面部肖像比其他题材更能打动人，还因为我们太擅长观察脸，所以肖像画家不得不加倍努力才能达到我们的要求。如果画的是树，树枝画偏了五度也不会有人发现。但是，如果你把别人的眼睛画偏了五度，人们一眼就能看出来。

德国包豪斯（Bauhaus）学派的设计师采纳了美国建筑师路易斯·沙利文（Louis Sullivan）的观点“功能决定形式”（form follows function），但是他们实际上的理解是“功能应当决定形式”。[①]真实情况是，如果开发“功能”非常艰难，那么“形式”将不得不全部都由“功能”决定，因为没有多余的精力再来单独开发“形式”了。人们常常觉得野生动物非常优美，原因就是它们的生活非常艰苦，在外形上不可能有多余的部分了。



好设计是看似容易的设计。优秀运动员比赛时，让人觉得他轻轻松松就获胜了，优秀设计师也是如此，他们的工作看上去很容易。大多数时候，这是一种错觉。作家的文章读起来流畅自如，但是背后其实经过了反复修改。

科学和工程学的一些最重大的发现在形式上往往很简单，会使得你觉得自己也想到过。可是，如果它真的那么简单，为什么发现人不是你呢？

达·芬奇的有些肖像画只是几根线条。看着它们，你会想只要把这

① 沙利文的原话是“功能总是决定形式”（form ever follows function），所以如今的引用形式实际上不准确，但是我觉得误读后的形式更接近于现代主义建筑师的观点。

十根八根线条放对位置，你也能画出如此优美的肖像画。说的没错，可是难就难在找出正确的位置。只要位置偏移一点点，整幅作品就会一溃千里。

白描其实是最难画的视觉媒介，因为它们要求几近完美的再现。用数学语言说，线条属于闭合解（closed-form solution），水平不够的艺术家没有办法直接解决问题，只能通过不断逼近来求解。许多孩子在十岁左右放弃了绘画，原因之一就是这时他们开始学习成年人的绘画技法，首先练习用线条勾勒出人脸。

在大多数领域，看上去容易的事情，背后都需要大量的练习。练习的作用也许是训练你把刻意为之的事情变成一种自觉的行为。有时，我们的训练只是为了让身体养成下意识的反应。优秀钢琴家弹奏名曲可以不经过大脑直接完成，艺术家也是这样，熟练以后，脑海中的艺术形象会自动从手上流淌出来，仿佛有人在一旁为他打节奏一样。



人们有时会说自己有了“状态”，我的理解是，他们这时可以控制自己的脊髓。脊髓是更本能的反应，面对难题时，它能释放你的直觉。

好设计是对称的设计。对称也许只是简洁性的一种表现，但是它十分重要，值得单独列为一点。自然界的对称大量存在，这就说明了对称的重要性。

对称有两种：重复性对称和递归性对称。递归性对称就是指子元素的重复，比如树叶上叶脉的纹路。

历史上，对称曾经泛滥一时，导致现在它在某些领域已经不流行了。从维多利亚女王时代开始，建筑师就有意多建造不对称的建筑。20 世纪 20 年代，不对称成了现代主义建筑的一个明确的前提条件。但是即使如此，这些建筑物往往也只是在主轴上不对称，细节部分依然大量使用对称。

在写作中，你会发现对称无处不在，短语、句子、小说的情节都是如此。音乐和美术也大量使用对称。拼接式的美术作品（还有塞尚的一部分作品）有非常强烈的视觉感染力，原因就是整幅作品由相同的作图元素构成，这也属于对称。对称性构图产生了一些最让人难忘的绘画作

品，尤其是那些两个半边互相呼应的作品，比如米开朗基罗的壁画《创世纪》和格兰特·伍德的油画《美国式哥特》。

在数学和工程学中，递归尤其有用。归纳式证明方法既简洁又美妙。在软件中，能用递归解决的问题通常代表已经找到了最佳解法。巴黎的埃菲尔铁塔如此引人注目，部分原因就是它的外形是递归的，大塔上面还有小塔（图 9-2）。

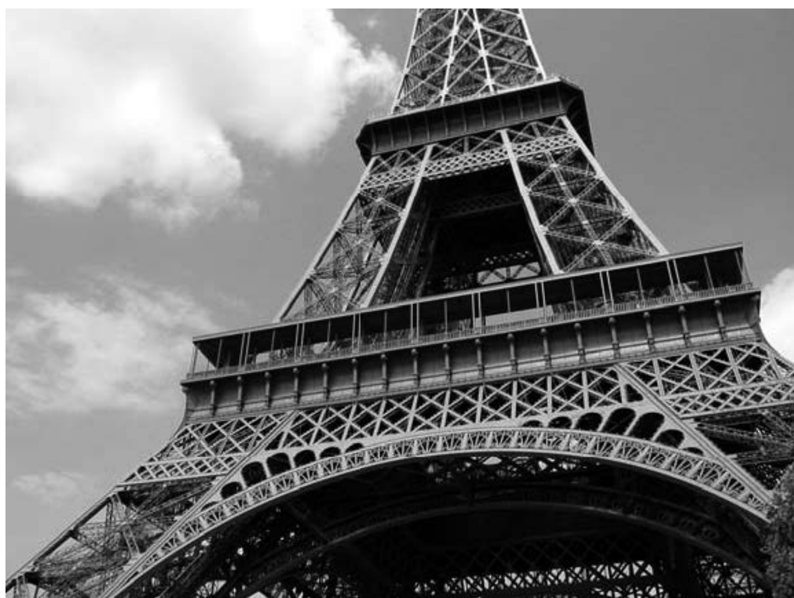


图 9-2　埃菲尔铁塔，1889。大塔上面有小塔



对称的危险在于它可以用来取代思考，在大量使用重复的时候这种危险性更大。

好设计是模仿大自然的设计。我不是说模仿大自然这种行为本身有多么好，而是说大自然在长期的演化中已经解决了很多设计问题。所以，如果你的设计与大自然很接近，那么它基本上不会很差。

模仿与剽窃并不相同。如果一部小说写得好像真实生活的再现，没人会提出异议。虽然写实的价值常常被误解，但它也是绘画的一个重要工具。写实的目的不是为了给生活留下一模一样的记录，而是为你的思想提供一个咀嚼点：你的眼睛看着某样东西，你的手就代表你的思想，

画出一些比较有意思的内容。

模仿大自然也是工程学的有效方法。长久以来，船只就像动物一样有龙骨和肋骨。不过，前提条件是技术水平要达到，只有这样才有可能模仿大自然。早期的飞机设计师按照鸟的形状设计飞机，这样做其实是错的，因为那时还没有足以模拟鸟类行为的轻型材料和能源，也做不出高度复杂的控制系统，所以飞机还不可能像鸟类那样飞。[①]但是，我能想象五十年后，小型的无人侦察飞机可以做得完全像鸟一样。

现在的计算机已经很强大了，不仅能模拟出大自然的环境，还能模拟大自然发展演变的结果。遗传算法可能会创造出正常条件下难以设计的复杂事物。





图 9-3　达·芬奇，《一匹直立的马的研究》，1481～1499

好设计是一种再设计。很少有人一次就把事情做对。专家的做法是

① 莱特兄弟的飞机引擎大约重70公斤，动力为12马力。F-18战斗机的F414-GE-400引擎重1.1吨，推动力接近10吨。换算后可知，两者的单位重量引擎产生的动力相差114倍。如今英特尔处理器的计算能力大约是30年前的1700倍。

先完成一个早期原型，然后提出修改计划，最后把早期原型扔掉。

扔掉早期原型是需要信心的，你必须有本事看出什么地方还可以改进。举例来说，刚刚开始学画的人往往不愿意重画画错的地方。他们觉得能画成现在这样已经很不错了，如果重画某些部分，结果可能还不如现在。所以，他们就说服自己，我的画已经过得去了，没准别人也会这么看。

这想法很危险。你应该培养对自己的不满。达·芬奇为了把一根线画对，经常要画五六次。保时捷 911 型汽车的原型很粗糙，只有在重新设计后它的背部轮廓才变成现在这样独特的曲线。建筑师莱特设计的古根海姆博物馆，最早的时候，右半边有点像古代的塔庙（ziggurat），他后来把它倒过来，就成了现在的样子。

犯错误是很正常的事情。你不要把犯错看成灾难，要勇于承认、勇于改正。达·芬奇实际上重新发明了素描这种艺术形式，把它当作一种探索更多可能的方式。开源软件因为公开承认自己会有bug，反而使得代码的bug比较少。



做修改的时候，有一个合适的工具会使得改动更容易。美术史上，15 世纪油彩取代蛋彩①（tempera）就是一个重大突破，油彩使得画家更方便地处理那些困难的主题（比如人体），因为油彩可以调制，还可以重画，蛋彩就做不到这些。

好设计是能够复制的设计。我们对待复制的态度经常是一个否定之否定的过程。刚入门的新手不知不觉地模仿他人，逐渐熟练之后才开始创作原创性作品。最后他会意识到，把事情做对比原创更重要。

不知不觉的模仿几乎必然将导致坏设计。如果你不知道自己的想法从何而来，那么你可能就是在模仿另一个模仿者。19 世纪中期，拉斐尔画派主导了整个画坛，几乎每个学画的人都在模仿拉斐尔，可是经常谬以千里。有一些艺术家实在看不下去了，被如此之多模仿拉斐尔的人搞

① 蛋彩画是15世纪的欧洲绘画方式，盛行于文艺复兴初期，主要是将鸡蛋和水作为溶剂，溶解绘画颜料，使之可以用来绘画。——译者注

烦了，于是成立了前拉斐尔画派[①]。

等到你逐渐对一件事产生热情的时候，就不会满足于模仿了。你的品味就进入了第二阶段，开始自觉地进行原创。

我想，最伟大的大师最终会达到一种超脱自我的境界。他们一心想找到正确答案，如果别人已经回答出了一部分，那就没理由不拿来用。他们足够自信地使用他人的成果，完全不担心因此丧失个人的特点。

好设计常常是奇特的设计。某些最出色的作品堪称不可思议：欧拉公式、16 世纪画家勃鲁盖尔的《雪中猎人》（图 9-4）、SR-71“黑鸟”超音速侦察机（图 9-5）、计算机的 Lisp 语言等。它们不仅优美，而且美得很奇特。





图 9-4　勃鲁盖尔的《雪中猎人》，1565 年

① 前拉斐尔画派是1848年由三个年轻的英国画家创立的。他们认为，学院派的方法已经腐化了米开朗基罗和拉斐尔的风格，因此艺术发展的正确道路应该回到拉斐尔之前的古典时代。——译者注



图 9-5　洛克希德公司的 SR-71“黑鸟”超音速侦察机，1964 年

我不太确定原因，可能是因为我不够聪明，才会觉得它们看上去很奇特。一条狗看到开罐器也会认为那是一个奇迹。如果我是天才的话，可能会觉得 $e^{i\pi}=-1$ 是再平常不过的事情，它又没有说错，有什么好奇怪的。

我在前文提到的好设计的大多数特点都是可以培育出来的，但是我觉得“奇特”这个特点是无法培育的。你最多就是在它开始显现时不要把它扼杀掉。爱因斯坦并不想让相对论变得很奇特，他只想找出真理，是真理本身显得很奇特。

我曾在一家美术学校学习绘画，那里的学生最想做的就是发展出一种自己的风格。但是，如果你想做出好作品，不可避免地会采用一种独特的方式，就好像每个人走路的姿势其实都不尽相同。米开朗基罗并没想过要树立米开朗基罗风格，他只是想画好作品，结果不由自主地创造出了米开朗基罗风格。

你最后发展出来的风格是自然而然形成的。“奇特”这个特点尤其如此，没有其他路可走。它就像连接大西洋和太平洋的“西北航道”，无数人希望找到这条捷径。16 世纪的风格主义者、19 世纪的浪漫主义者、一代代的美国高中生都在寻找，但就是找不到。唯一达到“奇特”的方法，就是追求做出好作品，完成之后再回过头看。

好设计是成批出现的。15 世纪住在佛罗伦萨的伟大艺术家有建筑师



布鲁内莱斯基、画家吉贝尔蒂、雕塑家多纳泰洛、画家马萨乔、画家菲利普里皮、画家弗拉安吉利科、雕塑家韦罗基奥、画家波提切利、达·芬奇和米开朗基罗。当时，米兰也是同等的大城市，请问你能说出 15 世纪米兰城有什么伟大艺术家吗？

15 世纪的佛罗伦萨有一些独特的条件，它们是不可延续的，因为今天的佛罗伦萨已经不是如此了。我们还必须假设达·芬奇和米开朗基罗拥有的天赋，在米兰城里一定也有人拥有。那么为什么没有出现米兰的达·芬奇呢？

今天，生活在美国的人口大概是 15 世纪佛罗伦萨的一千倍。那么按照比例推算，在我们之中存在着一千个达·芬奇和一千个米开朗基罗。如果这种推算成立，我们应该每天都看到令人惊叹的艺术奇迹。但是，事实并非如此，原因就是达·芬奇的出现除了他本身的天赋以外，还有赖于 1450 年的佛罗伦萨。



推动人才成批涌现的最大因素就是，让有天赋的人聚在一起，共同解决某个难题。互相激励比天赋更重要，达·芬奇之所以成为达·芬奇，主要原因不仅仅是他的天赋，更重要的是他生活在当时的佛罗伦萨，而不是米兰。今天，人类生活的流动性高得多，但是伟大的项目依然不成比例地集中在少数几个热点上：德国包豪斯建筑学院、曼哈顿计划、《纽约人》杂志、洛克希德公司的臭鼬工作室、施乐公司的帕洛阿尔托研究中心。

在历史的任何时刻都有一些热点项目，一些团体在这些项目上做出伟大的成绩。如果你远离这些中心，几乎不可能单靠自己就取得伟大成果。某种程度上，你个人最多可以对趋势产生一定的影响，但是你不可能决定趋势，实际上是趋势决定了你。（或许有人办得到，但是米兰的达·芬奇显然没有办到。）

好设计常常是大胆的设计。在任何一段历史中，人们都会把某些荒谬的东西当作正确的，并且深信不疑，以至于一旦你出言质疑，就有被排挤或者被暴力伤害的危险。

我们自己的这个时代要是不同以往，当然令人欢欣鼓舞。但是就我

所知，它并没有任何不同。

这个问题不仅存在于每个年代，还或多或少存在于每个领域。许多文艺复兴时期的艺术作品在当时都被认为极其大逆不道。根据意大利画家瓦萨里的记载，波提切利因此向教会忏悔并且放弃绘画，巴尔托洛梅奥和洛伦索迪克雷迪则是把自己的作品烧掉。爱因斯坦的相对论触犯了许多同时代的物理学家，许多年后还没有被完全接受，法国物理学家直到 20 世纪 50 年代才接受相对论。[①]

今天的实验性错误就是明天的新理论。如果你想做出伟大的新成果，那就不能对常识与真理不相吻合之处视而不见，反而应该特别注意才对。

实际上，我觉得发现丑陋的东西要比你想象出一个优美的东西更容易。大多数做出优美成果的人好像只是为了修正他们眼中丑陋的东西。伟大成果的出现常常来源于某人看到一样东西后，心想我能做得比这更好。拜占庭帝国的《圣母像》最早是根据某个公认的模板画的，非常机械呆板。几百年后的 14 世纪，意大利画家乔托看到以后，深感不满，决定动手改进，他因此成为文艺复兴的先行者。哥白尼对地心说无法解释的事情深感困扰，他的同时代人都觉得这可以忍受，他却认为一定能找到一种更好的解释。

单单是无法容忍丑陋的东西还不够，只有对这个领域非常熟悉，你才可能发现哪些地方可以动手改进。你必须锻炼自己。只有在成为某个领域的专家之后，你才会听到心里有一个细微的声音说：“这样解决太糟糕了！一定有更好的选择。”不要忽视这种声音，要培育它们。优秀作品的秘诀就是：非常严格的品味，再加上实现这种品味的能力。



① 参见Stephen G. Brush所著的“为什么相对论被接受了？”，*Physics in Perspective*，1999年第1期。

10

编程语言解析

所有机器都有一张操作命令清单，让你可以控制它。有时这个清单非常简短。电水壶就只允许两种操作：打开和关闭。CD播放器稍微复杂点，除了打开和关闭以外，还能调节音量、播放、暂停、快进、快退、随机播放等。

计算机和其他机器一样，也有一张操作命令清单。比如，可以命令计算机把两个数相加。这种操作命令的总和就是计算机的机器语言（machine language）。



机器语言

计算机刚发明的时候，所有程序就是一条条机器语言的命令。没过多久，程序就改成使用汇编语言了，它要比机器语言写起来稍微方便一点。命令清单还是一样的，就是每个命令换了一个更人性化的名字。机器语言的加法命令是11001101，这可能就是计算机内部的加法表达方式，但是在汇编语言中，这条命令就改成了add。

机器语言和汇编语言的共同问题就是，只能让大多数计算机做一些很简单的事情。比如，假定你想让计算机的蜂鸣器响10次，但是不存在一条直接的机器语言命令让电脑重复进行 n 次操作，所以只能用机器语言写出下面这样的程序：

```
a  将数字 10 存入内存地址 0
   如果内存地址 0 的值为负数，跳到 b 行
   蜂鸣器发出声音
   将内存地址 0 的值减 1
```

```
    跳到 a 行
b  ……程序的其他部分……
```

如果只是为了让蜂鸣器响 10 次就不得不写这么多代码，不难想象写出一个文字处理器或电子表格将是一项多么浩大的工程。

顺便说一句，请再看一下上面的程序。蜂鸣器真的会响 10 次吗？不，响了 11 次。我不应该在第一行使用 10，而应该使用 9。我故意在这个例子中留了一个bug，证明编程语言的一个重要特点：一个操作所需的代码越多，就越难避免bug，也越难发现它们。

高级语言

现在假设你不得不用汇编语言开发程序，但是你有了一个助手，他可以帮你承担那些麻烦的脏活。所以，你只要把程序写成下面这样就行了[①]：

```
dotimes 10 蜂鸣器响
```



接下来，你的助手会用汇编语言来实现这条命令（假定他不会产生bug）。

事实上大多数程序员就是这样工作的，不同之处就是，程序员的助手不是一个人，而是*编译器*。所谓“编译器”，本身就是一个程序，作用是将简便方式书写的程序（就像上面这一行命令）转变为硬件可以理解的语言。

这种简便方式书写的程序所使用的语言就叫做*高级语言*。它让你能够使用更强大的命令开发程序，比如现在你就有了“重复 n 次操作”的命令，不再仅限于只能做简单的“两个数相加”。

写程序时有了方便的命令，就可以把程序写得更简短。在上面假想的例子中，高级语言写出来的程序的长度只有机器语言的五分之一。所以，要是你犯错了，现在也更容易发现。

高级语言还有一个优点，它使得程序更具有可移植性。不同计算机的机器语言都不是完全相同的。所以，你无法将为某一种机型写的机器语言程序放到另一种机型上运行，只有彻底重写才能实现。但是，如果

① dotimes是Lisp语言中表示循环处理的命令。——译者注

你的程序是用高级语言写的，你只需要重写编译器就可以了。

编译器不是高级语言唯一的实现方法，另一种方法是使用*解释器*，它的作用是实时地将代码解释为相应的机器语言，然后一行行运行。相比之下，编译器则是先将整个程序全部翻译成机器语言，然后再运行。

开放源码

编译器处理的高级语言代码又叫做*源码*。它经过翻译以后产生的机器码就叫做*目标码*。顾客购买市场上的商业软件时得到的往往只是目标码。（目标码很难读懂，所以相当于被加密了，可以保护公司的商业秘密。）但是，后来出现另一种潮流：开放源码的软件。你可以得到源码，并且可以不受限制地修改它。

这两种方式的真正区别在于，开放源码使你对软件有更大的控制权，如果你想理解开源软件如何运行，只要阅读源码就行了。如果愿意，你甚至可以修改软件、重新编译。



你之所以需要这样做，一个原因可能是为了修正bug。比如，你自己不可能修正Windows的bug，因为你没有源码。（理论上你也许可以破解目标码，但是实际上这是非常难的。另一方面，软件的授权协议一般也不允许你这样做。）这会导致很大的问题。一旦Windows出现新的安全漏洞，只能等待微软公司发布解决方法，这还算是快的。如果bug的危害性不严重，只是偶尔会让你的机器死机，那么可能不得不等到下一次全面升级后问题才会得到解决。

开放源码的优势还不仅局限于可以自己动手解决bug。这里的关键是所有人都可以参与。所以，开源软件就像一篇经受同行评议的论文。许许多多的聪明人仔细阅读了Linux和FreeBSD这样的开源操作系统的源码，发现并且解决了大量的bug。相比之下，Windows的可靠性只能依赖于大公司自己的质量保证部门了。

开放源码的拥护者常常被看作反对知识产权的怪人。其中有些人确实如此，但是我本人肯定不反对知识产权。只是如果你要我安装没有源码的软件，我会非常犹豫。普通的消费者也许不需要看到他们使用的文字处理器的源码，但是在非常强调软件可靠性的情况下，出于强烈的工

程需求的考虑，会要求开放源码。

语言的战争

绝大多数程序员在绝大多数时候都使用高级语言编程。现在很少有人使用汇编语言。程序员的时间要比计算机的时间昂贵得多，后者已经变得很便宜了，所以几乎不值得非常麻烦地用汇编语言开发软件。只有少数最关键的部分可能还会用到汇编语言，比如开发某个计算机游戏时，你需要在微观层面控制硬件，使得游戏速度得到最大限度的终极提高。

Fortran、Lisp、Cobol、Basic、C、Pascal、Smalltalk、C++、Java、Perl 和 Python，全都是高级语言。它们只是比较出名的几种而已。现在的高级语言大概有几百种之多。不同机器语言的指令集基本相同，但是高级语言就不一样，它们开发程序的模式差别相当大。

那么，应该使用哪一种语言？嗯，关于这个问题，现在有很多争论。部分原因是，如果你长期使用某种语言，你就会慢慢按照这种语言的思维模式进行思考。所以，后来当你遇到其他任何一种有重大差异的语言，即使那种语言本身并没有任何不对的地方，你也会觉得它极其难用。缺乏经验的程序员对于各种语言优缺点的判断经常被这种心态误导。

可能因为想炫耀自己见多识广，某些黑客会告诉你所有高级语言基本相似。“所有编程语言我都用过。”某个看上去饱经风霜又酷的黑客往酒吧里一坐，“你用什么语言并不重要，重要的是你对问题是否有正确的理解。代码以外的东西才是关键。”

这当然是一派胡言。各种语言简直是天差地别，比如 Fortran I 和最新版的 Perl 就是两种完全不同的语言，而早期版的 Perl 和最新版的 Perl 之间的差别也大得惊人。但是，那个夸夸其谈的黑客可能真的相信自己的这番话，的确有可能使用所有不同的语言写出了与用原始的 Pascal 语言写的差不多的程序。如果你吃过麦当劳，就会知道全世界各地的麦当劳的味道都几乎一样。

一些黑客只喜欢自己用的语言，反感其他所有的语言。另一些黑客则说所有的语言都一样。事实介于这两个极端之间。语言之间确实有差别，但是很难确定地说哪一种语言是最好的。这个领域依然还在快速发展。



抽　象　性

高级语言比汇编语言更接近人类语言，而某些高级语言又比其他语言更进一步。举例来说，C 语言是一种低层次语言，很接近硬件，几乎堪称可移植的汇编语言，而 Lisp 语言的层次则是相当高。

如果高层级语言比汇编语言更有利于编程，你也许会认为语言的层次越高越好。一般情况下确实如此，但不是绝对的。编程语言可以变得很抽象，完全脱离硬件，但也有可能走错了方向。比如，我觉得 Prolog 语言就有这个问题。它的抽象能力强得不可思议，但是只能用来解决 2% 的问题，其余时间你苦思冥想、运用这些抽象能力写出来的程序实际上就是 Pascal 语言的程序。

另一个你会用到低层次语言的原因就是效率问题。如果你非常关注运行速度，那么最好使用接近机器的语言。大多数操作系统都是用 C 语言写的，这并非偶然。不过，硬件的运行速度越来越快了，所以使用 C 这样的低层次语言开发应用程序的必要性正在不断减少，但是大家似乎还是要求操作系统越快越好。（另一种可能是，人们还是希望“缓存区溢出攻击”继续存在下去，以便让大家时时保持警惕。[①]）



安全带还是手铐?

语言设计者之间的最大分歧也许就在于，有些人认为编程语言应该

① 最常见的几种入侵计算机的手法都是利用了C语言的某些特点。当你在C语言中为输入的内容分配出一片内存（也叫“缓存”）时，它会被分配在当前运行代码的返回地址旁边。所谓“返回地址”指的是一块特定内存，当前代码运行完毕以后，就要运行这块内存中包含的代码。也就是说，它实际上是计算机下一步要做的事情。
假定有人打算入侵你的计算机，他们猜出你会为某种输入分配256字节的缓存，于是他们就提交多于256字节的内容，目的是覆盖旁边的“返回地址”。那么，当前代码运行完毕之后，程序的控制权就交给了他们指定的内存地址。这个地址通常是缓存的首地址，缓存中是入侵者事前编好的机器码。于是，入侵者的程序就运行在你的计算机上了。
如果使用更抽象的高级语言，上面的事情是不可能发生的。但是，在C语言中，一旦接受用户输入的时候你没有检查输入长度，就创造出了一个安全漏洞。利用这种漏洞的攻击行为就被称为“缓冲区溢出攻击”。在这种攻击中，还有其他方法可以控制计算机，但是覆盖返回地址是最经典的一种。
有意思的是，劫持飞机与“缓冲区溢出攻击”有类似之处。在一般飞机上，乘客区与驾驶舱是相通的，就好像C语言中数据区与代码区是相邻的一样。劫机者一旦进入驾驶舱，实际上就相当于把自己从数据提升为代码。

防止程序员干蠢事，另一些人则认为程序员应该可以用编程语言干一切他们想干的事。Java 语言是前一个阵营的代表，Perl 语言则是后一个阵营的代表。（美国国防部很看中 Java 也就不足为奇了。）

自由语言派的信徒嘲笑另一方是“B&D”（奴役和戒律，Bondage and Discipline）语言，很无礼地暗示用那些语言编程的人是下等人。我不知道对方如何反击这些喜欢 Perl 的自由派，也许他们不喜欢给别人起绰号，因此我就无从知道。

由于防止程序员做蠢事有好几种方法，所以上面的争论逐渐分化成几个较小的议题。目前最活跃的议题之一就是*静态类型语言*与*动态类型语言*之争。在静态类型语言中，写代码时必须知道每个变量的类型。而在动态类型语言中，随便什么时候，你都可以把变量设为任意类型的值。

静态类型语言的拥护者认为这样可以防止bug，并且帮助编译器生成更快的代码（这两点理由都成立）。动态类型语言的拥护者认为静态类型对程序构成了限制（这点理由也成立）。我本人更喜欢动态类型，痛恨那些限制我的自由的语言。但是，确实有一些很聪明的人看来喜欢用静态类型语言。所以，这个问题依然值得讨论，并没有固定答案。



面向对象编程

眼下另一个争论的热点则是*面向对象编程*。它是一种不同的组织程序的方法。假定你要写一个程序，计算二维图形的面积。首先，你必须知道到底是圆形还是正方形。一种解决方法是用一整块的代码判断遇到的是什么图形，然后再用相应的公式计算面积。面向对象编程不是这样，它的方法是写出两个*类*，一个是圆形类，另一个是正方形类，然后每个类里面用一小块代码（叫做*方法*）计算该类图形的面积。求面积的时候，你就问要用哪一个类，然后再使用相应的方法得出最后答案。

这两种不同的计算方法可能听上去很相似，事实上，运行代码后，实际计算面积的*运算过程*也很相似。（这不奇怪，因为你本来就在解决同一个问题。）但是，代码的形式却是大相径庭。在面向对象编程的方式中，计算圆面积和正方形面积的代码可能分散在不同的文件中。与圆形有关

的代码都放在一个文件中，与正方形有关的代码则放在另一个文件中。

面向对象编程的优点在于，如果你需要修改程序，计算另一种图形的面积，比如三角形，你只需要再另外增加一块相应的代码就可以了，甚至可以不修改程序的其他部分。但是，批评者会反驳说，这种方法的缺点是，由于增加代码不用考虑其他部分，结果往往导致写出性能不佳甚至有副作用的代码，就好比造房子不考虑已经完成的部分一样。

关于面向对象编程优劣的争论并不像静态类型与动态类型之争那样壁垒分明，因为编程的时候你只能在静态类型和动态类型之中选一种。但是，面向对象编程只是程度不同的问题。事实上有两种程度的面向对象编程：某些语言允许你以这种风格编程，另一些语言则强迫你一定要这样编程。

我觉得后一类语言不可取。允许你做某事的语言肯定不差于强迫你做某事的语言。所以，至少在这方面我们可以得到明确的结论：你应该使用允许你面向对象编程的语言。至于你最后到底用不用则是另外一个问题了。



文艺复兴

有一件事，我想所有软件业的人都会同意，那就是最近出现了很多新的编程语言。直到 20 世纪 80 年代，只有大机构才买得起开发编程语言所需的硬件，所以大多数编程语言都是大公司的教授或者研究员开发的。而现在，一个高中生就能搞到所有必需的硬件。

Perl 语言的设计者拉里 · 瓦尔[①]的例子启发了很多黑客：为什么不动手设计一种自己的语言呢？只要你懂得驾驭开源软件社区，就会有很多人在短期内为你提供大量的代码。

结果就是产生了一些也许可以称为“头重脚轻”的语言：它们的内核设计得并非很好，但是却有着无数强大的函数库，可以用来解决特定的问题。（你可以想象一辆本身性能很差的小汽车，车顶却绑着一个飞机发动机。）有一些很琐碎、很普遍的问题，程序员本来要花大量时间来解

① Larry Wall（1954—）在大学里主修语言学。1987年为了使管理机房的工作变得方便，他在业余时间创造了Perl语言。——译者注

决，但是有了这些函数库以后，解决起来就变得很容易，所以这些库本身可能比核心的语言还要重要。所以，这些奇特组合的语言还是蛮有用的，一时间变得相当流行。车顶上绑着飞机发动机的小车也许真能开，只要你不尝试拐弯，可能就不会出问题。[①]

另一个结果就是语言的多样化。编程语言之间总是存在很大区别。Fortran、Lisp、APL 都是 1970 年以前开发出来的，它们之间的区别大得就像海星、熊、蜻蜓之间的区别。新兴的开源编程语言肯定将继承这种传统。

现在好像每隔一段日子就能听到一种新出现的语言。乔纳森·埃里克森把这种现象称为“编程语言的文艺复兴”。人们有时还会用另一个说法，即“编程语言的战争”。这并不矛盾，文艺复兴时期就是存在很多战争的。

实际上，很多历史学家相信战争是文艺复兴的一个副产品。[②]当时，欧洲活力旺盛可能就是因为它分成许多互相竞争的小国。它们互相毗邻，所以新思想能够从一个国家传播到另一个国家，但是它们又互相独立，使得单个的统治者无法遏制创新的发展。相比之下，中国古代的封建皇朝禁止民间建造大型的远洋船只，阻止了经济的正常发展。

所以，程序员活在这个文艺复兴时代可能是一件好事。如果我们所有人都使用同一种编程语言，反而有可能是坏事。



① 提醒各位亲爱的黑客，我只是打一个比方，请不要尝试在车顶绑上飞机发动机。另外，可以认为这类“头重脚轻”的语言存在已久，Fortran语言的流行主要就是因为它的函数库。

② 参见Carlo Cipolla所著的《枪，帆船，帝国：技术革新在1400~1700年欧洲扩张早期阶段的作用》（*Guns, Sails, and Empires: Technological Innovation and the Early Phases of European Expansion 1400-1700*），Pantheon，1965年出版。

11

一百年后的编程语言

很难预测一百年后的人类生活，只有少数几件事是可以确定的。那时，汽车将具备低空飞行能力，城市规划的法规将放宽，大楼可以造到几百层，大街上一天到晚看不见太阳，女性个个都学过防身术。本文只想讨论其中的一个细节：一百年后，人们使用什么语言开发软件？

为什么这个问题值得思考？原因不是我们最终会用上这些语言，而是幸运的话，我们从现在开始就能用上这些语言。



我认为，编程语言就像生物物种一样，存在一个进化的脉络，许许多多分支最终都会成为进化的死胡同。这种现象已经发生了。Cobol 语言曾经流行一时，但是现在看来没有任何后续语言继承它的思想。它就像尼安德特人[①]一样，进化之路已经走到了尽头。

我预言Java也会如此。有人写信说："你怎么能说Java不会成功呢？它已经成功了。"我觉得这要看你的成功标准是什么。如果标准是相关书籍的出版量，或者是相信学会Java就能找到工作的大学生数量，那么Java确实已经成功了。当我说Java不会成功时，我的意思是它和Cobol一样，进化之路已经走到了尽头。

这只是我的猜测，未必正确。这里的重点不是看衰 Java，而是提出编程语言存在一个进化的脉络，从而引导读者思考，在整个进化过程中，某一种语言的位置到底在哪里？之所以要问这个问题，不是为了一百年后让后人感叹我们曾经如此英明，而是为了找到进化的主干。它会启发

---

① 尼安德特人（Neanderthal），一种生活在欧洲的古人类，三万多年前已经全部灭绝。
——译者注

我们去选择那些靠近主干的语言，这样对当前的编程最有利。

无论何时，选择进化的主干可能都是最佳方案。要是你不幸选错了，变成了一个尼安德特人，那就太糟了。你的对手克鲁马努人时不时就会来攻打你，把你的食物全部偷走。

这就是我想找出一百年后的编程语言的原因。我不愿意押错赌注。

编程语言的进化与生物学进化还是有区别的，因为不同分支的语言会发生聚合。比如，Fortran分支看来正在与Algol[①]的继承者聚合。理论上，不同的生物物种也可能发生聚合，但是可能性很低，所以大概从来没有真正出现过。

编程语言之所以可能出现聚合，一个原因是它的概率空间[②]比较小，另一个原因是它的突变不是随机的。语言的设计者们总是有意识地借鉴其他语言的设计思想。



对于语言设计者来说，认清编程语言的进化路径特别有用，因为这样就可以照着样子设计语言了。这时，认清进化的主干就不仅有助于识别现存的优秀语言，还可以把它当作设计语言的指南。

任何一种编程语言都可以分成两大组成部分：基本运算符的集合（扮演公理的角色）以及除运算符以外的其他部分（原则上，这个部分可以用基本运算符表达出来）。

我认为，基本运算符是一种语言能否长期存在的最重要因素。其他因素都不是决定性的。这有点像买房子的时候你应该先考虑地理位置。别的地方将来出问题都有办法弥补，但是地理位置是没法变的。

慎重选择公理还不够，还必须控制它的规模。数学家总是觉得公理越少越好，我觉得他们说到了点子上。

你仔细审视一种语言的内核，考虑哪些部分可以被摒弃，这至少也

① Algol语言诞生于20世纪50年代，是最早的计算机语言之一，对后来的许多语言产生了极大的影响。——译者注

② 概率空间是一个数学术语，大致指概率的可能取值范围。这里的意思是，不管编程语言怎么变，它的形式总是很有限的。——译者注

是一种很有用的训练。在长期的职业生涯中，我发现冗余的代码会导致更多冗余的代码，不仅软件如此，而且像我这样性格懒散的人，我发现在床底下和房间的角落里这个命题也成立，一件垃圾会产生更多的垃圾。

我的判断是，那些内核最小、最干净的编程语言才会存在于进化的主干上。一种语言的内核设计得越小、越干净，它的生命力就越顽强。

当然，猜测一百年后人们使用什么编程语言，这本身就是一个很大的假设。也许一百年后人类已经不编程了，或者直接告诉计算机想做什么，计算机就会自动完成。

不过，到目前为止，计算机智能并没有取得太大进展。我猜测一百年后，人们还是使用与现在差不多的程序指挥计算机。可能有一些我们今天需要编程解决的问题，那时已经不需要编程了，但是我想，那时还会存在大量与今天一样的编程任务。



你可能认为只有那些自以为是的人才会去预言一百年后的技术。但是，请不要忘记，软件发展的历史已经走过了 50 年。在这 50 年中，编程语言的进化其实是非常缓慢的，因此展望一百年后的语言并不是虚无缥缈的想法。

编程语言进化缓慢的原因在于它们并不是真正的技术。语言只是一种书写法，而程序则是一种严格符合规则的描述，以书面形式记录计算机应该如何解决你的问题。所以，编程语言的进化速度更像数学符号的进化速度，而不像真正的技术（比如交通或通信技术）的进化速度。数学符号的进化是缓慢的渐变式变化，而不是真正技术的那种跳跃式发展。

无论一百年后的计算机是什么样子，我们基本上可以断定它们的运行速度一定会快得多。如果摩尔定律依然成立，一百年后计算机的运行速度将是现在的74乘以10的18次方倍（准确地说是73 786 976 294 838 206 464倍）。真是让人难以想象。不过实际上更现实的预测并不是速度会提高这么多，而是摩尔定律最终将不成立。不管是什么东西，如果每18个月就增长一倍，那么最后很可能会达到极限。但那时的计算机比现在快得多

大概是毫无疑问的。即使最后只是略微快了100万倍，也将实质性地改变编程的基本规则。如果其他条件不变，现在被认为运行速度慢的语言（即运行的效率不高）将来会有更大的发展空间。

那时，依然会有对运行速度要求很高的应用程序。我们希望计算机解决的有些问题其实是计算机本身引起的。比如，计算机处理视频的速度取决于生成这些视频的另一台计算机。此外，还有一些问题本身就要求无限快的处理能力，比如图像渲染、加密/解密、模拟运算等。

既然在现实中一些应用程序本身的效率较低，而另一些应用程序会耗尽硬件提供的所有运算能力，那么有了更快速的计算机就意味着编程语言不得不应付更多的极端情况，涵盖更大范围的效率要求。我们已经看到这种情况发生了。要是以几十年前的标准衡量，有一些使用新语言开发的热门应用程序对硬件资源的浪费非常惊人。

不仅编程语言有这种现象，这实际上是一种普遍的历史趋势。随着技术的发展，每一代人都在做上一代人觉得很浪费的事情。30 年前的人要是看到我们今天如此随意地使用长途电话，一定会感到震惊。100 年前的人要是看到一个普通的包裹竟然也能享受一天内从波士顿发件、途经孟菲斯、抵达纽约的待遇，恐怕就要更震惊了。



我已经预测了，一旦未来硬件的性能大幅提高将会发生什么事。新增加的运算能力都会被糟蹋掉。

在我学习编程的年代，计算机还是稀罕玩意。我记得当时使用的微机型号是 TRS-80，它的内存只有 4K，为了把 BASIC 程序装入内存，我不得不把源码中的空格全部删除。我一想到那些极其低效率的软件，不断重复某些愚蠢的运算，把硬件的计算能力全部占用，就感到无法忍受。但是，我的这种反应是错的，我就像某个出身贫寒的穷孩子，一听到要花钱就舍不得，即使把钱用在重要场合（比如去医院看病）都觉得很难接受。

某些浪费确实令人厌恶。比如有人就很讨厌 SUV（运动型多用途车），即使它采用可再生的清洁能源也改变不了看法，因为 SUV 来自一个令人厌恶的想法（如何使得小货车看上去更有男子汉气概）。但是，并非所有

的浪费都是坏的。既然如今的电信基础设施已经如此发达，再掐着时间打长途电话就有点锱铢必较了。如果有足够的资源，你可以将长途电话和本地电话视为同一件事，一切会变得更轻松。

浪费可以分成好的浪费和坏的浪费。我感兴趣的是好的浪费，即用更多的钱得到更简单的设计。所以，问题就变成了如何才能充分利用新硬件更强大的性能最有利地“浪费”它们？

对速度的追求是人类内心深处根深蒂固的欲望。当你看着计算机这个小玩意，就会不由自主地希望程序运行得越快越好，真的要下一番功夫才能把这种欲望克制住。设计编程语言的时候，我们应该有意识地问自己，什么时候可以放弃一些性能，换来一点点便利性的提高。

很多数据结构存在的原因都与计算机的速度有关。比如，今天的许多语言都同时有字符串和列表。从语义上看，字符串或多或少可以理解成列表的一个子集，其中的每一个元素都是字符。那么，为什么还需要把字符串单列为一种数据类型呢？完全可以不这么做。只是为了提高效率，所以字符串才会存在。但是，这种以加快运行速度为目的、却使得编程语言的语义大大复杂的行为，很不可取。编程语言设置字符串似乎就是一个过早优化的例子。



如果我们把一种语言的内核设想为一些基本公理的集合，那么仅仅为了提高效率就往内核添加多余的公理，却没有带来表达能力的提升，这肯定是一件很糟的事。没错，效率是很重要，但是我认为修改语言设计并不是提高效率的正确方法。

正确做法应该是将语言的语义与语言的实现予以分离。在语义上不需要同时存在列表和字符串，单单列表就够了。而在实现上做好编译器优化，使它在必要时把字符串作为连续字节的形式处理。①

对于大多数程序，速度不是最关键的因素，所以你通常不需要费心

① 我相信，Lisp Machine Lisp（Lisp语言的一种方言）是第一个具体表达这样一种观点的语言：变量的声明（除了动态类型变量之外）只是优化的建议，对一个正确程序本身的含义不构成影响。Common Lisp（Lisp语言的另一种方言）则好像第一个明确提出了这一点。

考虑这种硬件层面上的微观管理。随着计算机速度越来越快，这一点已经越发明显了。

语言设计时，对实现方式少作限制还会使得程序具备更大的灵活性。语言的规格发生变化不仅是无法避免的，也是合理的。通过编译器的处理，按照以前规格开发的软件就会照常运行，这就提供了灵活性。

essay（论文）这个词来自法语的动词 essayer，意思是“试试看”。从这个原始意义来说，论文就是你写一篇文章，试着搞清楚某件事。软件也是如此。我觉得一些最好的软件就像论文一样，也就是说，当作者真正开始动手写这些软件的时候，他们其实不知道最后会写出什么结果。

Lisp 语言的黑客早就明白数据结构灵活性的价值。我们写程序的第一版时，往往会把所有事情都用列表的形式处理。所以，这些最初版本可能效率低下得惊人，你必须努力克制自己才能忍住不动手优化它们，这就好像吃牛排的时候必须努力克制自己才能不去想牛排是从哪里来的一样，至少对我来说是这样的。

一百年后的程序员最需要的编程语言就是可以让你毫不费力地写出程序第一版的编程语言，哪怕它的效率低下得惊人（至少按我们今天的眼光来看是如此）。他们会说，他们想要的就是很容易上手的编程语言。

效率低下的软件并不等于很烂的软件。一种让程序员做无用功的语言才真正称得上很烂。浪费程序员的时间而不是浪费机器的时间才是真正的无效率。随着计算机速度越来越快，这会变得越来越明显。

我觉得，放弃字符串类型已经是大家可以接受的想法了。Arc语言已经这样做了，看上去效果不错。以前用正则表达式很难描述的一些操作，现在用回归函数可以表达得很简单。

这种数据结构的扁平化趋势会怎么发展？我极其努力地设想各种可能，得到的结果甚至令我自己都吓了一跳。比如，数组会不会消失？毕竟数组只是散列表的一个子集，其特点就是数组的键全部都是整数向量。进一步说，散列表本身会不会被列表取代呢？

还有比这更惊人的预言。在逻辑上其实不需要对整数设置单独的表



示法，因为可以把它们也看作列表，整数 n 可以用一个 n 元素的列表表示。这一样能完成数学运算，只是效率低得让人无法忍受。

编程语言会发展到放弃基本数据类型之一的整数这一步吗？我这样问并不是真的要你严肃思考这个问题，更多的是希望打开你对未来的思路。我只是提出一种假想的情况：如果一股不可抗拒的力量遇到了一个不可移动的物体，会发生什么事。具体就本文而言：一种效率低得不可想象的语言遇到了性能强大得不可想象的硬件，会发生什么事。我看不出放弃整数类型有什么不妥。未来相当漫长。如果我们想要减少语言内核中基本公理的数目，不妨把眼光放得远一点，想一想如果时间变量 t 趋向无限会怎么样。一百年是一个很好的参考指标，如果你觉得某个想法在一百年后仍然可能是难以令人接受，那么也许一千年后它也依然难以令人接受。



让我说清楚，我的意思不是说所有的整数运算都用列表来实现，而是说语言的内核（不涉及任何编译器的实现）可以这样定义。在现实中，任何进行数学运算的程序可能都是以二进制形式表示数字，但是这属于编译器的优化，而不属于语言内核语义的一部分。

另一种消耗硬件性能的方法就是，在应用软件与硬件之间设置很多的软件层。这也是我们已经看到的一种趋势，许多新兴的语言就被编译成字节码[①]。比尔·伍兹曾经对我说，根据经验判断，每增加一个解释层，软件的运行速度就会慢一个数量级。但是，多余的软件层可以让编程灵活起来。

Arc 语言[②]最初的版本就是一个极端的例子，它的层很多，运行速度非常慢，但是确实带来了相应的好处。Arc 是一个典型的“元循环”（metacircular）解释器，在 Common Lisp 的基础上开发，很像约翰·麦卡锡在他经典的 Lisp 论文中定义的 eval 函数。Arc 解释器一共只有几百

① 字节码（byte code）是已经经过编译但是需要进一步处理才能变成机器码的中间代码。它的好处是与硬件和软件环境无关，在编译器的配合下，可以在不同的操作系统上运行。字节码的典型运用就是Java语言。——译者注

② Arc是Lisp的一种方言，由本书作者提出，目前由他本人和罗伯特·莫里斯负责开发。——译者注

行代码，所以很便于理解和修改。我们采用的 Common Lisp 版本是 CLisp，它本身是在另一个字节码解释器的基础上开发的。所以，我们一共有两层解释器，最上面那层效率低下得惊人，但是语言本身是能用的。我承认只是勉强可用，但是确实能用。

即使是应用程序，使用多层形式开发也是一种很强大的技巧。自下而上的编程方法意味着要把软件分成好几层，每一层都可以充当它上面那一层的开发语言。这种方法往往会产生更小、更灵活的程序。它也是通往软件圣杯——可重用性（reusability）——的最佳路线。从定义上看，语言就是可以重用的。在编程语言的帮助下，你的应用程序越是采用这种多层形式开发，它的可重用性就越好。

可重用性这个概念多多少少与 20 世纪 80 年代兴起的面向对象编程有些关联。不管怎样寻找证据，也不可能把这两件事完全分开。某些使用面向对象编程开发出来的软件确实具有可重用性，但是这不是因为它使用了面向对象编程，而是因为它的开发方法是自下而上的。以函数库为例，它们具有可重用性，是因为它们属于语言的一部分，而不是因为它们采用面向对象或者其他编程方法。

顺便说一句，我不认为面向对象编程将来会消亡。我觉得，除了某些特定的领域，这种编程方法其实没有为优秀程序员带来很多好处，但是它对大公司有不可抗拒的吸引力。面向对象编程使得你有办法对面条式代码进行可持续性开发。通过不断地打补丁，它让你将软件一步步做大。大公司总是倾向于采用这样的方式开发软件。我预计一百年后也是如此。

既然是谈论未来，最好谈谈并行计算（parallel computation），因为看上去并行计算好像就是为未来而存在的。无论怎么想，并行计算似乎都是未来生活的一部分。

它会在未来实现吗？过去二十年，人们都在说并行计算马上就会来临。但是，到目前为止，它对编程实践并没有太大影响。这是真的吗？芯片设计师已经不得不把它考虑在内，为多 CPU 计算机开发系统软件的



程序员也是如此。

但是，真正的问题在于，并行计算到底能达到哪个抽象层次？一百年后它就会影响到开发应用软件的程序员吗？或者，它还只是编译器作者需要考虑的事情，在应用软件的代码中根本就无处寻觅？

一种可能是，大多数可以用到并行计算的场合，人们都会放弃使用并行计算。虽然我总的预测是未来的软件会挥霍掉大部分新增的硬件性能，但是并行计算是一个特例。我估计随着硬件性能得到惊人的提升，如果你明确地说想要并行计算，那么肯定可以得到它，但是通常情况下你不会用到它。这意味着，除了一些特殊的应用程序，一百年后的并行计算不会是那种大规模的并行计算（massive parallelism）。我预料，对于普通程序员来说，一切更像对进程进行复制，然后让多个进程在后台并行运行。



这是编程进行到很后期才要做的事情，属于对程序的优化，类似于你想开发一种特定的数据结构来取代现有的数据结构。程序的第一个版本通常会忽略并行计算提供的各种好处，就好像编程开始时会忽略某种特定的数据结构给你带来的好处一样。

除了某些特定的应用软件，一百年后，并行计算不会很流行。如果应用软件真的大量使用并行计算，这就属于过早优化了。

一百年后会有多少种编程语言？从最近来看，出现了大量的新语言。硬件性能提高是一个原因，这就允许程序员根据使用目的在运行速度和编程便利性之间做出不同的取舍。如果这就是未来的趋势，那么一百年后强大的硬件只会使得语言数目变得更多。

但是，另一方面，一百年后的常用语言可能只有很少几种。部分原因是基于我的乐观主义，我相信在未来，如果你的作品确实很出色，你可能选择的是一种开发起来很方便的语言。使用这种语言写出来的软件第一版的运行速度很慢，只有对编译器进行优化设置后运行速度才会提升。既然我抱有这种乐观主义，那么我还要做一个预言。有些语言可以达到机器的最高效率，另一些语言的效率则慢到刚刚可以运行而已，两者之间存在巨大的差距。我预言一百年后，这段差距之间的各个点上都

会有对应的编程语言存在。

因为这段差距正在变得越来越大，所以性能分析器（profiler）将变得越来越重要。目前，性能分析并没有受到重视。许多人好像仍然相信，程序运行速度提升的关键在于开发出能够生成更快速代码的编译器。代码效率与机器性能的差距正在不断加大，我们将会越来越清楚地看到，应用软件运行速度提升的关键在于有一个好的性能分析器帮助指导程序开发。

我说将来可能只有很少几种常用语言，但没有把用于特定领域的“小众语言”（little language）算进去。我觉得，这些嵌入式语言的想法很不错，一定会蓬勃发展。但是我判断这些“小众语言”会被设计成相当薄的一层，使得用户可以一眼看出在底下作为基础的通用型语言，这样就减少了学习时间，降低了使用成本。

谁来设计这些未来的语言？过去10年最激动人心的趋势之一就是开源语言的崛起，比如 Perl、Python 和 Ruby。语言设计已经被黑客接管。到目前为止这样到底是好是坏还看不清楚，但是发展势头令人鼓舞。比如，Perl 就有一些绝妙的创新。不过，它也包含了一些很糟糕的想法。对于一种充满进取心、大胆探索的语言来说，这也是很正常的事。以它现在这种变化的速率，大概只有上帝才知道一百年后 Perl 会变成什么样。

有一句俗话说，如果你自己做不到，那就去当老师。这在语言设计领域不成立，我认识的一些最出色的黑客就在当教授。但是，当老师的人确实有很多事情不能做。研究性职位给黑客带来了一些限制。在任何学术领域，都有一些题目是可以做的，另一些题目是不可以做的。不幸的是，这两类题目的区别通常取决于它们写成论文后看上去是不是很高深，而不是取决于它们对软件业的发展是否重要。最极端的例子可能就是文学，文学研究者的任何成果几乎对文学创作者都毫无影响。

虽然科学领域的状况要稍好一点，但是研究者可以做的题目与能够对设计优秀语言有所帮助的题目之间的交集小得令人沮丧。（奥林· 希弗斯曾经对这一点表达不满，而且说得头头是道。）比如，研究变量类型的论文好像多得无穷无尽，尽管事实上静态类型语言看来无法真正支持宏（在我看来，一种语言不支持宏，那就不值得使用了）。



新语言更多地以开源项目的形式出现，而不是以研究性项目的形式出现。这是语言的一种发展趋势。另一种发展趋势是，新语言的设计者更多的是本身就需要使用它们的应用软件作者，而不是编译器作者。这似乎是好的趋势，我期待它继续保持下去。

一百年后的物理学基本上不可能预测。但是计算机语言不一样，现在就动手设计一种一百年后可以吸引使用者的新语言，这在理论上似乎是可能的。

设计新语言的方法之一就是直接写下你想写的程序，不管编译器是否存在，也不管有没有支持它的硬件。这就是假设存在无限的资源供你支配。不管是今天还是一百年后，这样的假设好像都是有道理的。

你应该写什么程序？随便什么，只要能让你最省力地写出来就行。但是要注意，这必须是在你的思维没有被当前使用的编程语言影响的情况下。这种影响无处不在，必须很努力才能克服。你也许觉得，对于人类这样懒惰的生物，喜欢用最省力的方式写程序是再自然不过的事情。但是事实上，我们的思想可能往往会受限于某种现存的语言，只采用在这种语言看来更简单的形式，它对我们思想的束缚作用会大得令人震惊。新语言必须靠你自己去发现，不能依靠那些让你自然而然就沉下去的思维定势。



采用程序的长度作为它耗费工作量的近似指标是个很有用的技巧。这里的程序长度当然不是指字符的数量，而是指各种句法元素的总长度，基本上就是整个解析树的大小。也许不能说最短的程序就是写起来最省力的程序，但是当你一心想把程序写得简洁而不是松松垮垮时，你就更接近省力这个目标，你的日子也会变得好过得多。所以，设计语言的正确做法就变成了，看着一段程序，然后问自己是不是能把它写得更短一点？

实际上，用想象出来的一种一百年后的语言来写程序，这件事情的可靠程度，取决于你对语言内核的估计是否足够正确。常规的排序，你现在就可以写出来。但是，想要预测一百年后的语言使用什么函数库就

很难了。很可能许多函数库针对的领域现在还根本不存在。比如，如果SETI@home[①]计划成功，我们就需要与外星人联系的函数库了。当然，如果外星人的文明高度发达，已经到了用XML格式交换信息的地步，那就不需要新的函数库了。

另一个极端是，我觉得今天你就能设计出一百年后的语言内核。事实上，在有些人看来，大部分语言内核在 1958 年就已经设计出来了。[②]

如果今天就能使用一百年后的编程语言，我们会用它编程吗？观古而知今。如果1960年就能使用今天的编程语言，那时的人们会用它们吗？

在某些方面，回答是否定的。今天的编程语言依赖的硬件在 1960 年并不存在。比如，Python 这样的语言，正确的缩进（indentation）在编写时很重要，但是 1960 年的计算机没有显示器，只有打印机终端，所以编写起来就不会很顺利。但是，如果把这些因素排除在外（你可以假设，我们只在纸上编程），20 世纪 60 年代的程序员会喜欢用现在的语言编程吗？



我想他们会的。某些缺乏想象力、深受早期编程语言思想影响的人可能会觉得不可能。（没有指针运算，如何复制数据？没有goto语句，如何实现流程图？）但是我想，那时最聪明的程序员一定能轻松地使用今天的大多数语言，假定他们能得到的话。

如果我们现在就能拥有一百年后的编程语言，那就至少能用来写出优秀的伪码[③]。我们会用它开发软件吗？因为一百年后的编程语言需要为某些应用程序生成快速代码，所以很可能它生成的代码能够在我们的硬件上运行，速度也还可以接受。相比一百年后的用户，我们也许不得不

---

① SETI@home是一个寻找地球以外智慧生命的科学实验，由加州大学伯克利分校发起并主持。它使用射电望远镜监听太空中的无线电信号，然后用计算机进行数据分析，如果发现有些信号不可能自然产生，就可以证明外星文明的存在。1995年，该项目决定向志愿者开放，使用全球联网的大量计算机进行分布式计算，1999年5月开始正式运行。详细情况参见http://setiathome.berkeley.edu。——译者注

② Lisp语言的第一版规格说明书是1958年发布的。——译者注

③ 伪码又称虚拟代码，用来抽象地描述算法，而不是现实存在的编程代码。——译者注

对这种语言做更多的优化，但是总的来看，它应该仍然会为我们带来净收益。

现在，我们的两个观点就是：（1）一百年后的编程语言在理论上今天就能设计出来；（2）如果今天真能设计出这样一种语言，很可能现在就适合编程，并且能够产生更好的结果。如果我们把这两个观点联系起来，那就得出了一些有趣的可能性。为什么不现在就动手尝试写出一百年后的编程语言呢？

当你设计语言的时候，心里牢牢记住这个目标是有好处的。学习开车的时候，一个需要记住的原则就是要把车开直，不是通过将车身对齐画在地上的分隔线，而是通过瞄准远处的某个点。即使你的目标只在几米开外，这样做也是正确的。我认为，设计编程语言时，我们也应该这样做。



12

拒绝平庸

1995年，罗伯特·莫里斯和我一起创办了Viaweb。我们打算开发软件，让用户可以自己搭建网上商店。当时，我们的创意是把软件放在服务器端，使用普通的网页作为用户界面。

当然，那个时候许多人可能都想到过这个主意。但是，就我所知，Viaweb 是第一个互联网应用程序。在我们看来，这真的是很新颖的想法，所以我们就把公司命名为 Viaweb①，意即我们的软件通过网络使用，而不是运行在你的桌面电脑上。

另一个特别之处是，这个软件主要采用 Lisp 语言开发。②它是最早的用 Lisp 语言开发的大型应用程序，在此之前，Lisp 语言主要用于大学和实验室中。



秘密武器

埃里克·雷蒙德写过一篇文章《如何成为一名黑客》（How to Become a Hacker）。文中有一部分专门谈到，在他看来，如果你想当一个黑客，应该学习哪些语言。他建议从 Python 和 Java 入手，因为它们比较容易学。想当高级一点的黑客，还应该学习 C 和 Perl。前者用来对付 Unix 系统，

---

① 在英语中，via是一个介词，意为“经过……”，所以Viaweb的意思就是经过网络。
——译者注

② 一开始的时候，Viaweb有两个部分——编辑器和订单处理系统。前者用Common Lisp开发，主要供用户搭建自己的网站。后者用C语言开发，主要用来处理订单。在Viaweb的第一版中，Lisp是最主要的开发语言，因为订单处理系统非常小，占用的代码很少。2003年1月，Yahoo发布了Viaweb编辑器的新版本，采用C++和Perl开发。但是，为了把原始程序翻译成C++，他们可能不得不专门写一个Lisp解释器，因为据我所知，Viaweb所有的页面生成模板还没变，都是使用Lisp代码。（参见Greenspun写的*Tenth Rule*一书第198页。）

后者用来系统管理和开发 CGI 脚本。最后，真正非常严肃地把黑客作为人生目标的人，应该考虑学习 Lisp：

> Lisp很值得学习。你掌握它以后，会感到它给你带来的极大启发。这会大大提高你的编程水平，使你成为一个更好的程序员。尽管在实际工作中极少会用到Lisp。

在讨论学习拉丁语有何价值时，你往往也会听到这一类的话。拉丁语无助于你找工作（也许古典文学教授的工作除外），但是它可以训练你的思维，帮助你更好地运用母语（比如英语）进行写作。

但是且慢，拉丁语的比喻并不完全适合 Lisp 语言。拉丁语无助于你找工作的原因是因为没有人说拉丁语。如果你用它写作，没有人能看懂。但是，Lisp 是一种计算机语言，无论我们程序员使用哪一种语言与计算机交谈，它都能听懂。



如果埃里克·雷蒙德没有说错，Lisp 语言确实可以使你成为更好的程序员，那么为什么你不使用它编程呢？如果画家有一支让他画得更好的画笔，我觉得他应该会用这支笔完成所有的画作，对不对？我在这里不是想证明埃里克·雷蒙德错了。他的观点整体上非常正确，他对 Lisp 语言的看法确实是大多数人的看法，但是这里面就是有一个矛盾：Lisp 语言能让你成为更好的程序员，但你却不用它，这难道不奇怪吗？

为什么不用呢？编程语言毕竟是一种工具。如果 Lisp 语言真的能开发出更好的程序，你就应该用它。如果它无助于编程，那么就不会有人需要它。

这不仅仅是一个理论问题。软件业是竞争非常激烈的行业，而且容易出现垄断。在不考虑其他情况的条件下，某家公司的软件更快更好用，就会把竞争者赶出这个市场。一旦你开始创业，你就会更深切地感受到这一点。一般情况是，创业公司要么赢得一切，要么彻底失败。你要么成为富翁，要么一无所获。创业的时候，如果你选择了错误的技术，竞争对手就会一举打败你。

罗伯特·莫里斯和我都很了解 Lisp 语言，我们相信自己的直觉，找不出任何不使用它的理由。我们知道其他人都用 C++或 Perl 开发软件，

但是我们不觉得这说明了什么问题。如果别人用什么技术，你也用什么技术，那么你大概只能使用 Windows 了。选择使用哪一种技术的时候，你不能考虑别人的做法，只能考虑什么样的技术能最好地完成工作。



图 12-1　我和罗伯特·莫里斯在 Viaweb，1996 年年初



创业公司尤其如此。大公司可以互相模仿，但是创业公司就不行。我觉得很多人没有意识到这一点，尤其是一些创业者。

大公司每年平均成长大约 10%。所以，如果你掌管一家大公司，只要每件事都做到大公司的平均水准，你就能得到大公司的平均结果，也就是每年成长大约 10%。

如果你掌管创业公司，当然也可以这样。你把每件事都做到平均水准，就能得到平均结果。问题在于，小公司的平均结果就意味着关门倒闭。创业公司的生存率远低于 50%。所以，如果你掌管创业公司，最好做一些独特的事情，否则就会有麻烦。

回到 1995 年，我们懂得一些竞争对手不懂的事情（至少在我们看来是如此），这些事情甚至直到今天都很少有人懂：如果开发只在自己服务器上运行的软件，这意味着你想用什么语言就能用什么语言。如果开发桌面软件，就完全不一样了，大多数情况下你只能使用操作系统所用的开发语言。10 年前，开发桌面软件就意味着要使用 C 语言。但是，对于互联网软件，你能使用任何你想用的语言。如果你还同时拥有操作系统和语言的源码，那么你的自由就更大了。

但是，这种新出现的自由是一把双刃剑。既然你可以使用任何语言，你就不得不思考到底使用哪一种语言。如果你的公司对这种选择的自由视而不见，而竞争对手看到了，那么你就有被击败的危险。

如果选择哪种语言都行，你到底使用哪一种语言？我们选择 Lisp。首先，很明显，对于这个市场来说，快速开发出产品是很重要的。我们所有人都是从零开始，所以能够快速做出新功能的公司就会取得巨大的竞争优势。我们知道 Lisp 语言真的非常合适快速开发软件，而且我们的软件运行在服务器端，你一写完代码就能发布出去，所以这又进一步放大了快速开发的效果。

如果其他公司不想使用 Lisp 语言，那就更好了。这会让我们拥有技术优势。我们不能放过任何有利的因素。创办 Viaweb 的时候，我们对于如何经营一家公司毫无经验，对市场推广、雇用员工、融资、发展新客户等都一无所知。在此之前，我和莫里斯甚至连一天正式上班的经历都没有。我们唯一擅长的事情就是开发软件。我们希望这一点可以弥补我们的劣势。任何在软件开发上面有助于我们获得优势的事情我们都不能放过。



可以这样说，我们使用 Lisp 只是一个大胆的冒险。我们设想如果用 Lisp 语言开发自己的软件，就能比竞争对手更快地写出新功能，还能做到他们做不到的事情。同时，因为 Lisp 是一种抽象层次非常高的语言，所以就不需要非常庞大的开发团队，这会降低成本。如果我们的设想是正确的，那么我们就能用更少的钱做出一个更好的产品，从而获得利润。最终，我们将独占市场，竞争对手什么也得不到，到头来只能退出这个行业。我们当时心里就是这么盘算的。

这次冒险的结果如何？多少有点出人意料，它竟然达到了我们的设想。我们前前后后遭遇到很多竞争对手，一共大概有二三十个，但是他们的软件没有一个能与我们竞争。我们的软件运行在服务器端，用户可以“所见即所得”地搭建网上商店，感觉就像在操作桌面软件。我们的竞争对手使用 CGI 脚本。我们在功能上总是遥遥领先于他们。有时，他们出于绝望，试图引入我们没有的功能。但是，有了 Lisp 语言的帮助，我们的开发周期很短。有时候，竞争对手刚刚发布新闻稿宣布将引入新功能，我们就能在一两天内做出自己的版本。当对手找来的记者抽出时间打电话

过来想了解我们的反应，我们就会告诉他我们已经有了这个功能。

竞争对手一定觉得我们好像拥有了某种秘密武器，能够破解他们内部的通信或者其他机密。事实上，我们的确拥有秘密武器，但是没他们想的那么复杂。从来没有人向我们泄露他们的内部机密，只是我们的开发速度比别人想象的更快而已。

9岁时，我碰巧读过弗雷德里克·福赛思的小说《刺杀戴高乐》（*The Day of the Jackal*）。小说的主角是一个刺客，有人雇他暗杀法国总统。那个刺客必须通过警察的岗哨才能到达可以俯视总统行进路线的公寓。他扮成拄着拐杖的老头从警察身边经过，没有引起任何人的怀疑。

我们的秘密武器很类似上面的情景。我们使用一种奇特的人工智能语言开发软件，它的语法非常古怪，大量使用括号。多年来，要是听到别人这样描述Lisp语言，我会勃然大怒。但是现在，这却成了我们的优势。在竞争中，你的对手无法理解你的技术优势，这可是再宝贵不过了。商场如战场，对手摸不透你，你的胜算就增加了。



虽然有些令人难为情，但是我必须承认，就是因为这个原因，在Viaweb创业期间我从来没有公开谈论过Lisp语言。我们对新闻媒体闭口不谈Lisp，如果你在我们的网站上搜索Lisp，只会发现我在个人介绍中提到过两次，那是我写的两本关于Lisp的书。这是故意的，创业公司对竞争对手应该越保密越好。如果他们不知道（或者不关心）我们的软件用什么语言开发，我就要把这个秘密保持下去。[①]

最了解我们技术的人就是客户。他们不关心Viaweb用什么语言开发，但是发现它真的很好用。Viaweb可以让用户在几分钟内搭建起漂亮的网上商店。因此，主要通过口碑效应，我们得到了越来越多的新客户。1996年年底，我们支持的网上商店大约是70家。1997年，变成了500家。6个月后，雅虎收购我们的时候，我们有1070个用户。更名为Yahoo Store之后，这个软件继续主导市场，它是雅虎获利最丰厚的业务之一，用它搭建的商店成为“雅虎购物”（Yahoo Shopping）的基础。我在1999年离开了雅虎，所以不知道现在的准确用户数量，但是我上一次听到的

① 莫里斯觉得不用这么保密，因为即使竞争对手知道我们使用Lisp语言，对他们也不会有帮助：“如果他们真的聪明，早就已经在用Lisp编程了。”

数字是超过了 2 万。

Blub 困境

Lisp 语言到底好在什么地方？如果它真的这么好，为什么没有得到广泛使用呢？这种问题听起来有点像绕口令，但是实际上回答起来很简单。Lisp 语言的好处不在于它有一些狂热爱好者才明白的优点，而只在于它是目前最强大的编程语言。它没有得到广泛使用的原因就是因为编程语言不仅仅是技术，也是一种习惯性思维，非常难于改变。当然，上面两句话都需要进一步解释。

我先从一个争议极大的命题开始讲起：编程语言的编程能力有差异。

至少不会有人反对高级语言比机器语言更强大这一观点。今天的大多数程序员通常情况下都不会想用机器语言编程，而是使用一种高级语言，然后再让编译器帮你把它翻译成机器语言。这种观念甚至已经移植到了硬件，从 20 世纪 80 年代开始，硬件的指令集都是针对编译器而不是针对程序员设计的。



大家都知道，徒手用机器语言写出整个程序是一件很蠢的事。但是，把这个观点推广到一种更普遍的情况，知道的人就不多了。如果你有好几种语言可以选择，在不考虑其他因素的情况下，你不选择最强大的那种语言就是一件很蠢的事。[①]

上面这个观点有许多例外情况。如果在开发的程序必须与另一个程序紧密配合，那么可能最好还是使用后者的开发语言。如果你的程序只是要做一些很简单的事（比如整数运算或者位操作），那就不妨使用一种比较靠近机器的低层次语言，主要原因是这样运行起来会更快一些。如果你的程序很短，只是为了特定场合一次性使用，那么你最好根据自己

① 如果从图灵等价（Turing-equivalent）的角度来看，所有语言都是一样强大的，但是这对程序员没有意义。（没人想为图灵机编程。）程序员关心的那种强大也许很难正式定义，但是有一个办法可以解释，那就是有一些功能在一种语言中是内置的，但是在另一种语言中需要修改解释器才能做到，那么前者就比后者更强大。如果A语言有一个运算符，可以移除字符串中的空格，而B语言没有这个运算符，这可能不足以称A语言比B语言强大，因为你可以在B语言里写一个函数实现这个功能。但是，如果A语言支持某种高级功能（假定是递归），而B语言不支持，你就不可能通过自己编写函数库解决了，所以这就代表A语言比B语言更强大。

要解决的问题选择具有最强大函数库的语言。不过，总的来看，对于应用程序来说，还是应该选择总体最强大、效率也在可接受范围内的编程语言，否则都是不正确的选择，就好像你选择机器语言编程一样，只是程度上有差异而已。

大家都公认机器语言属于非常低层次的语言。但是，至少在社会上很多人眼里，高级语言其实也差不多。但事实并非如此，高级语言与机器语言的差别很大。从技术上看，“高级语言”并不是一个定义很清晰的名词。在高级语言与机器语言之间并不存在一条明确的分界线。语言的抽象性是一条连续曲线，[①]从最强大的语言一直到最底层的机器语言，每一种语言的能力都有差异。

以 Cobol 语言为例，通过编译器，它可以被编译成机器语言。从这个角度来说它是一种高级语言。但是，有谁会真的把 Cobol 当成与其他高级语言（比如 Python）一样强大的语言？比起 Python，它可能更接近机器语言。



Perl 4 如何？与 Perl 5 相比，它不支持闭包。所以，大多数 Perl 的黑客都认为 Perl 5 比 Perl 4 更强大。如果你同意这一点，就意味着你也认可一种高级语言可以比另一种高级语言更强大。因此，必然能够接着推导出，除了某些特殊情况，你就是应该使用目前最强大的语言。

不过在现实中这个结论很少能落实。到了一定年龄之后，程序员极少主动更换自己的编程语言。不管习惯使用的是哪一种语言，他们往往认为这种语言已经足够好了。

程序员非常忠于他们心爱的语言，我不想伤害任何人的感情，所以为了解释我的观点，我假设有一种 Blub 语言。它的抽象程度正好落在编程能力曲线的中点。它不是最强大的语言，但是要比 Cobol 或机器语言更高级。

我们假设 Blub 程序员既不使用机器语言也不使用 Cobol 语言。他认为前者是编译器的工作，后者他不知道有什么用（Cobol 语言甚至连 XX 功能也没有，Blub 语言就具备这个功能）。

---

① 语言之间的关系或许还可以比喻成栅格结构（lattice），从下往上朝着顶端慢慢收窄。具体的形状在这里并不重要，重点是语言之间至少存在着一种偏序关系（partial order）。

只要这位程序员向曲线下方望去，他就肯定知道自己正在看的是一些比较低层次的语言。因为那些语言明显不如Blub语言强大，缺少他习惯使用的某些功能。但是，当他向曲线上方望去，他不会意识到自己正在看更高层次的语言，而是仅仅觉得自己正在看某些奇怪的语言。他可能认为那些语言也许与Blub一样强大，但是加入了不少怪东西。他觉得Blub语言已经够用了，不用再考虑那些语言了。这时，他的思维就是已经被Blub同化了。

但是，当我们转换视角，把自己想象成使用曲线更上方某一种语言的程序员并往下看的时候，我们就会发现，自己也同样轻视Blub语言。你怎么用Blub语言完成工作呢？它甚至连YY功能都没有！

通过归纳法我们就会知道，唯一洞悉所有语言优劣的人必然是懂得最强大的那种语言的人。（这大概就是埃里克·雷蒙德所说的Lisp语言使你成为一个更好的程序员的意思。）由于Blub困境的存在，你无法信任其他人的意见：他们都满足于自己碰巧用熟了的那种语言，他们的编程思想都被那种语言主宰了。



我自己的经历也证实了这个看法。高中时我喜欢用Basic语言编程。这种语言功能很弱，甚至不支持递归，很难想象没有递归还怎么编程。但是我那时根本没觉得有损失，Basic语言控制了我的思维。当时我非常精通Basic语言，只要是学过的部分都能熟练地使用。

雷蒙德推荐的五种黑客应该学会的语言，其强大程度各有不同，分布在编程能力曲线五个不同的点上。它们的相对位置是一个敏感的话题。我只想说，我认为，Lisp语言在最上方。为了证明这个论断，让我告诉你，我发现Lisp有一个功能，其他四种语言都没有。我觉得，没有宏（macro）的话，那些语言怎么编程呢？[①]

许多语言自称也有宏，但是Lisp的宏是独一无二的。信不信由你，Lisp宏的作用与括号有关。Lisp语言的设计者大量使用括号并不是为了标新立异。Blub语言的程序员会觉得Lisp代码看上去很怪，有那么多括号，但这是有原因的。它们是Lisp与其他语言存在巨大差异的外在表现。

① 把宏说成一种独立的功能有误导之嫌。在实际运用中，如果没有其他Lisp功能（比如闭包和函数的rest参数）的配合，Lisp的宏也不会有太大作用。

Lisp 代码由 Lisp 数据对象构成。其他语言的源代码一般由字符组成，字符串是主要数据类型之一，但是 Lisp 语言不完全是这样。经过解析器处理之后，Lisp 代码就变成了你可以遍历的数据结构。

如果你理解编译器的工作原理，那么事实是，与其说 Lisp 有一种很奇特的语法，还不如说它根本没有语法。一般的源代码程序经过编译器解析会生成解析树。Lisp 的奇特之处就在于，你可以完全写出程序，控制这种解析树，进行任意的存取操作。Lisp 的这种程序就叫做宏，它们可以用来生成其他程序。

生成其他程序的程序？什么时候需要用到它们？如果你用 Cobol 语言思考，会觉得很少需要用到它们。如果你用 Lisp 语言思考，会发现它们无所不在。我要是在这里举一个 Lisp 宏功能强大的实例，可能更便于说明问题。你看这个例子！是不是很方便啊？但是如果这样做，对于不懂 Lisp 语言的人来说，这篇文章就不知所云了。本文没有办法把所有事情都解释清楚，无法帮助你彻底理解这门语言。我在 *Ansi Common Lisp* 一书中已经尽可能地简化内容、快速讲解，但是也要到全书篇幅将近一半的地方（第 11 章）才能讲到宏。



但是我想可以给出事实证明我的这个观点。Viaweb 编辑器的源码之中大约 20%~25%是宏。它们比普通的 Lisp 函数难写，而且如果用在不必要的地方，反而是一种很不良的编程习惯。所以，我们代码中的每一个宏都有充分的使用理由。这意味着这个程序至少 20%~25%代码的功能无法轻易地用其他语言实现。我在前文一再声称 Lisp 语言无比强大，无论 Blub 语言的程序员对此多么怀疑，看到这个事实应该足以让他感到很好奇，我们居然用到了这么多宏。我们这样写代码并不是为了好玩。我们是一家小创业公司，拼尽全力写代码，只是为了给竞争对手布下重重障碍，不让他们赶上来。

抱有怀疑态度的人可能会想上面的论断是否成立，两者之间是否存在相关关系。我们的一大块代码能够做到其他语言很难做到的事。只凭这一点是否能得出结论：我们的软件能够做到竞争对手的软件做不到的事？我必须说，这里面可能就是存在相关关系。我鼓励你继续深入思考这个问题。表面上，一个老年人拄着拐杖蹒跚而行，你不要只是看看而

已，他背后可能有更多的故事值得了解，你应该想得更深一些。

创业公司的合气道[①]

尽管 Lisp 语言非常强大，但是我并不期望有谁（超过 25 岁的人）读完这篇文章就立刻开始学习它。我写这篇文章的目的不是想改变任何人的观点，而是想让那些有兴趣学习 Lisp 语言的人放心，他们知道 Lisp 是一种强大的语言，但是担心使用者太少，学会了也没什么用。我想让他们明白，在商业竞争中使用 Lisp 语言就会带来优势。你的竞争对手不懂 Lisp，这将使得它的强大更充分地表现出来。

如果你想在创业公司中使用 Lisp 语言，你不仅不应该担心使用它的人太少，反而应该希望这种局面保持下去。事实上，现状很可能真的会保持下去。因为编程语言的特点之一就是它会使得大多数使用它的人满足于现状，不想改用其他语言。人类天性变化的速度大大慢于计算机硬件变化的速度，所以编程语言的发展通常比 CPU 的发展落后一二十年。在麻省理工学院这样的地方，20 世纪 60 年代初就开始使用高级语言了。但是，许多公司直到 80 年代还在用机器语言编程。我敢打赌，很多人对机器语言恋恋不舍，直到 CPU 开始采用精简指令集[②]了才不得不放弃使用机器语言。这就好比酒吧已经到了打烊时间，酒保开始整理桌子、收拾东西准备回家，客人才被迫离开。



技术的变化速度通常是很快的。但是，编程语言不一样，与其说它是技术，还不如说是程序员的思考模式。编程语言是技术和宗教的混合物。[③]所以，一种很普通的编程语言就是很普通的程序员使用的语言，它的变化就像冰山那样缓慢。大概在 1960 年，Lisp 语言引入了垃圾回收机

---

① 合气道（Aikido）是一种日本的武术，主要特点是“以柔克刚”、“借劲使力”、“不主动攻击”。——译者注

② 精简指令集计算机（Reduced Instruction Set Computer，简称RISC）是CPU的一种架构，对指令数目和寻址方式都做了精简，使其实现更容易，执行速度更快，编译器的效率更高。它在20世纪80年代开始得到大规模采用。——译者注

③ 所以，如果你想对编程语言进行比较，那就做好准备打一场宗教战争，或者索性就写一本绝对不带个人色彩的大学教材，枯燥得像人类学研究一样。那些喜欢平静生活的人以及想要得到终身教职的人对这个话题唯恐避之不及。但是，必须承认的是，它只是一半与宗教有关，所以剩下的一半依然值得研究，尤其是当你要设计新语言的时候。

制（Garbage Collection），今天已经被广泛认为是非常好的做法。Lisp 的动态类型特点也同样受到越来越多人的认同。闭包是 20 世纪 60 年代 Lisp 语言引入的功能，现在的接受程度还很低。宏也是 60 年代中期 Lisp 语言引入的，现在还是一片处女地。

很显然，那些很普通的编程语言正在主导一切。我不建议你挑战这种强大的习惯势力，相反，我建议你向日本合气道选手学习，利用这种势力削弱你的竞争对手，让他们自食其果。

如果你为大公司工作，想要改用 Lisp 语言可能不是一件容易的事。你很难说服自以为是的老板，让他允许你用 Lisp 语言开发程序。老板受到报纸的影响，认为某些其他语言将主宰世界（就像 20 年前 Ada 语言受到的评价）。但是，如果你为创业公司工作，那里没有这样的老板，那么你就能和我们一样，将他人的 Blub 困境转变为你的优势。你的竞争对手被牢牢粘在那些很普通的语言上面，永远都追不上你使用的技术。



如果你为创业公司工作，那么这里有一个评估竞争对手的妙招——关注他们的招聘职位。他们网站上的其他内容无非是一些陈腐的照片和夸夸其谈的文字，但是招聘职位却不得不写得很明确，反映出他们到底想干什么，否则就会引来一大批不合适的求职者。

在 Viaweb 创业期间，我读过大量竞争对手的招聘职位。差不多每个月都有一个新的竞争对手浮出水面。我首先会看他们的产品有没有一个试用版，然后就去找他们的招聘职位。这样过了几年，我就知道哪些公司值得关注，哪些公司不用在意。有些公司的职位描述使用了大量的 IT 词汇，这样的内容越多，这家公司就越不构成威胁。最不用担心的竞争对手就是那些要求应聘者具有 Oracle 数据库经验的公司，你永远不必担心他们。如果是招聘 C++或 Java 程序员的公司，对你也不会构成威胁。如果他们招聘 Perl 或 Python 程序员，就稍微有点威胁了。至少这听起来像一家技术公司，并且由黑客控制。如果我有幸见到一家招聘 Lisp 黑客的公司，就会真的感到如临大敌。

13

书呆子的复仇

软件业有一场永不停息的战斗，书生气的开发者与官僚主义的经理之间总是发生冲突。大家应该都看过漫画《呆伯特》，熟悉里面那个发型高耸的经理。[①]我想，技术行业的大部分人对这个角色都过目难忘，因为在他们的公司里就有这个角色的原型。

那些经理奇迹般地同时具备了两种很常见但很难结合在一起的特点：(a) 对技术一无所知；(b) 对技术有强烈的个人观点。



举例来说，假设你需要写一个软件。你的经理根本不懂这个软件的运作机制，也不知道各种编程语言有什么区别。但是，他竟然明确要求你一定要使用某一种语言进行开发。没错，他就是要求你一定要用 Java 语言。

为什么他会提出这种要求？让我们看看他脑袋里是怎么想的。他的想法无非就是，Java 是业界的标准。我知道肯定如此，因为媒体对此有铺天盖地的报道。既然它是标准，那么使用它就不会错。另外，这也意味着人才市场上肯定有无数 Java 程序员，即使现在为我打工的这批人都辞职了（真奇怪，这种事情总是不断发生），我也能够轻易地找到替代者。

嗯，这听起来也不无道理。但是，它的前提是一个没有说出口的假设，而这个假设实际上是错的。你的经理相信所有编程语言的功能都差不多，可以互相替代。如果这种想法是对的，那么他要求你用 Java 编程

---

① 这篇文章发上网后，我收到了一封显然发自肺腑的电子邮件，开头是这样写的：

> 发型高耸？难道有谁的头发不是向上耸的吗？如果你只能想出这种词去侮辱你的经理，那么你们这些人活该被称为“书呆子”。

就很合理了。反正编程语言之间没有区别，那么就用大家都在用的那种语言吧。

但是，编程语言是不一样的。就算不探讨各种语言之间的具体区别，我也能向你证明这一点。回到 1992 年，如果你问经理使用什么语言开发软件。他会像今天一样毫不迟疑地回答说 C++。如果所有编程语言都一样，为什么答案变了？进一步说，为什么 Java 语言的设计者要如此麻烦地去创造一种新语言呢？

一般来说，如果你动手创造一种新语言，那是因为你觉得它在某些方面会优于现有的语言。Java 语言之父詹姆斯·戈斯林在第一份《Java 白皮书》中说得很清楚，之所以要设计 Java，就是想解决 C++的一些弱点。所以结论就是，各种编程语言的编程能力是不相同的。如果你接受你的经理的假设，然后一路追溯到 Java 语言的源头，就会得到与他的假设完全不同的结果。



到底谁对？戈斯林还是你的经理？结果当然是意料之中的，戈斯林是正确的。某些情况下，一些语言就是比另一些语言更出色。可是这样一说又导致了另外的问题。C++不适合解决某些难题，所以 Java 才被设计出来。那么，什么情况下应该使用 Java，什么情况下应该使用 C++呢？会不会某些情况下其他语言比它们更合适呢？

一旦你开始思考这个问题，就会发现它非常棘手。如果你的经理被迫去想这个问题，当他看到它的复杂性时，脑袋恐怕都会爆炸。如果所有语言真的都一样，那么他只需选择一种看上去获得大部分人拥戴的语言就可以了，因为这实际上是一种流行风尚，而不是技术问题，所以即使像你的经理这样对技术无知的人也有可能轻松得到正确答案。但是，如果语言各有不同，你的经理就会突然发现，有两个互相关联的方程，他必须找到一个能够同时满足两个方程的最佳解，而最要命的却是他对此根本一无所知。第一个方程是找到（相对于要解决的问题）能够适用 20 年左右的最佳语言，第二个方程是（为这种语言）找到合适的程序员、函数库的机会有多大。如果假定所有语言都不同，就会遇到这种苦苦求解的情况，所以难怪你的经理不愿意接受这个假设了。

认为所有语言都一样的看法的缺点是自欺欺人，但是优点是可以使许多事情变得很简单。我想这就是为什么它被广泛接受的主要原因。它是一个令人舒服的想法。

大家都觉得 Java 一定有过人之处，因为它是一种很酷的新兴编程语言。但是真的如此吗？如果你站在远处观察编程语言的世界，似乎 Java 就是最新的东西。（如果你站得足够远，那么你看到的所有东西就是 Sun 公司出钱制作的大型霓虹广告牌。）但是，如果你靠近观察这个世界，就会发现不同的人对“酷”的理解是不一样的。在黑客圈子里，Perl 被公认比 Java 酷得多。黑客社区网站 Slashdot 就是用 Perl 开发的。我估计你不可能看到黑客愿意使用 Java 的 JSP 技术开发网站。可是，还有一种更新的语言叫做 Python，它的使用者往往看不起 Perl。另一些人则认为 Ruby 语言是取代 Python 的最佳选择。

当你按照 Java、Perl、Python、Ruby 这样的顺序观察这些语言，你会发现一个有趣的结果。至少，如果你是一个 Lisp 黑客，你就看得出来，排在越后面的语言越像 Lisp。Python 语言模仿 Lisp，甚至把许多 Lisp 黑客认为属于设计错误的功能也一起模仿了。至于 Ruby 语言，如果回到 1975 年，你声称它是一种有着自己句法的 Lisp 方言，没有人会提出反对意见。编程语言现在的发展不过刚刚赶上 1958 年 Lisp 语言的水平。



朝着数学的方法发展

1958 年，约翰·麦卡锡第一个提出了 Lisp 语言。我认为，当前最流行的编程语言不过只是实现了他在 1958 年的想法而已。

这怎么可能呢？计算机技术的发展不是日新月异吗？1958 年的计算机的运算能力还不如今天的电子表，而体积却大得像冰箱。[①]那时的技术怎么可能超过今天的水平呢？

---

① IBM 704型计算机的CPU就像冰箱一样大，并且重得多（1429千克）。4 K大小的RAM则装在另外一个箱子里，重达1800千克。相比之下，Sub-Zero 690是最大的家用冰箱型号之一，重量还不到300千克。



图 13-1　IBM 704，美国劳伦斯利弗莫尔国家实验室，1956 年



让我告诉你原因。这是因为设计者本来没打算把 Lisp 设计成编程语言，至少不是我们现在意义上的编程语言。我们今天所说的编程语言指的是用来告诉计算机怎么做的一种工具。麦卡锡最后确实有意开发这种意义上的编程语言，但是实际上他做出来的 Lisp 却是完全不同的一种东西，语言的基础是他的一种理论演算，他想用更简洁的方式定义图灵机。正如他后来所说：

> Lisp比图灵机表达起来更简洁。证明这一点的一种方法就是写一个Lisp通用函数，证明它比图灵机的一般性描述更短、更易懂。这个Lisp函数就是eval……它用来计算Lisp表达式的值……。编写eval函数需要发明一种表示法，能够把Lisp函数表示成Lisp数据。设计这种书写法完全是为了满足论文写作的需要。（我）根本没有想过用它来编写Lisp程序并在计算机上运行。

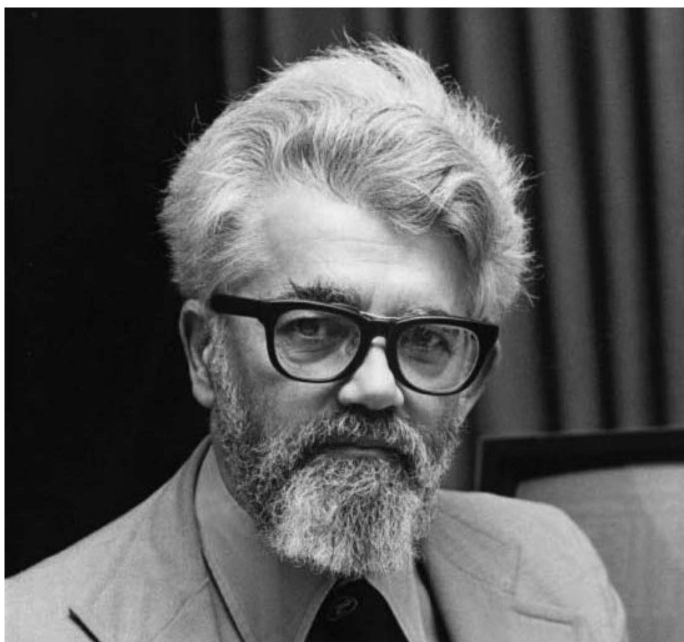


图 13-2　书呆子之王约翰·麦卡锡



1958 年年底，麦卡锡的一个学生史蒂夫·拉塞尔[①]看到了 eval 函数的定义，意识到如果把它翻译成机器语言，就可以把 Lisp 解释器做出来。

这在当时是非常令人吃惊的事。麦卡锡后来回忆：

> 拉塞尔对我说："我想把eval编成程序……"我告诉他，别把理论和实践混淆，eval只是用来读的，不是用来做计算的。但是他执意要做，并且还真的做出来了。就是说，他把我论文中的eval编译成了[IBM] 704计算机的机器码，修正了bug，然后对外宣布做出了Lisp语言的一种解释器，这倒没有说错，确实如此。所以，从那个时候开始，Lisp语言就基本上是它现在的样子了……

这样一下子，就在几个星期之内，麦卡锡发现他的理论演算变成了一种实际的编程语言，而且出乎意料地强大。

① Steve Russell，也是历史上第一个电脑游戏的作者，1962年他写了《太空大战》（*Spacewar*）。

由此也就得出了 20 世纪 50 年代的编程语言到现在还没有过时的原因。简单说，因为这种语言本质上不是一种技术，而是数学。数学是不会过时的。你不应该把 Lisp 语言与 50 年代的硬件联系在一起，而是应该把它与快速排序（Quicksort）算法进行类比。这种算法是 1960 年提出的，至今仍然是最快的通用排序方法。

Fortran 语言也是 20 世纪 50 年代出现的，并且一直使用至今。它代表了语言设计的一种完全不同的方向。Lisp 语言是无意中从纯理论发展为编程语言的，而 Fortran 从一开始就是作为编程语言设计出来的。但是，今天我们把 Lisp 看成高级语言，而把 Fortran 看成一种相当低层次的语言。

1956 年 Fortran 刚诞生的时候，叫做 Fortran I，与今天的 Fortran 语言差别极大。Fortran I 实际上是汇编语言加上数学，在某些方面还不如今天的汇编语言强大。比如，它没有子例程，只有分支跳转结构（branch）。今天的 Fortran 语言可以说更接近 Lisp 而不是 Fortran I。



Lisp和Fortran代表了编程语言发展的两大方向。前者的基础是数学，后者的基础是硬件架构。从那时起，这两大方向一直在互相靠拢。Lisp语言刚设计出来的时候就很强大，接下来的二十年它提高了运行速度。而那些所谓的主流语言把更快的运行速度作为设计的出发点，然后再用四十多年的时间一步步变得更强大。直到今天，最高级的主流语言也只是刚刚接近Lisp的水平。虽然已经很接近了，但还是没有Lisp那样强大。

为什么 Lisp 语言很特别

Lisp 语言诞生的时候就包含了 9 种新思想。其中一些我们今天已经习以为常，另一些则刚刚在其他高级语言中出现，至今还有 2 种是 Lisp 独有的。按照被大众接受的程度，这 9 种思想依次如下排列。

(1) 条件结构（即 if-then-else 结构）。现在大家都觉得这是理所当然的，但是 Fortran I 就没有这个结构，它只有基于底层机器指令的 goto 结构。

(2) 函数也是一种数据类型。在 Lisp 语言中，函数与整数或字符串一

样，也属于数据类型的一种。它有自己的字面表示形式（literal representation），能够存储在变量中，也能当作参数传递。一种数据类型应该有的功能，它都有。

(3) 递归。Lisp 是第一种支持递归函数的高级语言。①

(4) 变量的动态类型。在 Lisp 语言中，所有变量实际上都是指针，所指向的值有类型之分，而变量本身没有。复制变量就相当于复制指针，而不是复制它们指向的数据。

(5) 垃圾回收机制。

(6) 程序由表达式组成。Lisp 程序是一些表达式树的集合，每个表达式都返回一个值。这与 Fortran 和大多数后来的语言都截然不同，它们的程序由表达式和语句组成。

区分表达式和语句在 Fortran I 中是很自然的，因为它不支持语句嵌套。所以，如果你需要用数学式子计算一个值，那就只有用表达式返回这个值，没有其他语法结构可用，否则就无法处理这个值。



后来，新的编程语言支持块结构，这种限制当然也就不存在了。但是为时已晚，表达式和语句的区分已经根深蒂固。它从 Fortran 扩散到 Algol 语言，接着又扩散到它们两者的后继语言。

(7) 符号类型。符号实际上是一种指针，指向存储在散列表中的字符串。所以，比较两个符号是否相等，只要看它们的指针是否一样就行了，不用逐个字符地比较。

(8) 代码使用符号和常量组成的树形表示法。

(9) 无论什么时候，整个语言都是可用的。Lisp 并不真正区分读取期、编译期和运行期。你可以在读取期编译或运行代码，也可以在编译期读取或运行代码，还可以在运行期读取或者编译代码。

在读取期运行代码，使得用户可以重新调整（reprogram）Lisp 的语法；在编译期运行代码，则是 Lisp 宏的工作基础；在运行期编译代码，

① Lisp语言的许多特性（比如，把程序写成列表形式以及实现某种形式的递归）都在20世纪50年代的IPL-V语言中出现过。但是，IPL-V更像是汇编语言，它的程序中充满了操作码/地址对。参见Allen Newell等人编著的《IPL-V语言操作手册》（*Information Processing Language-V Manual*），Prentice-Hall，1961年出版。

使得Lisp可以在Emacs这样的程序中充当扩展语言（extension language）；在运行期读取代码，使得程序之间可以用S表达式（S-expression）通信，近来XML格式的出现使得这个概念被重新“发明”出来了。[①]

Lisp语言刚出现的时候，这些思想与其他编程语言大相径庭，后者的设计思想主要由50年代后期的硬件决定。随着时间流逝，流行的编程语言不断更新换代，语言设计思想逐渐向Lisp靠拢。思想(1)到思想(5)已经被广泛接受，思想(6)开始在主流编程语言中出现，思想(7)在Python语言中有所实现，不过似乎没有专用的语法。

思想(8)可能是最有意思的一点。它与思想(9)只是由于偶然原因才成为Lisp语言的一部分，因为它们不属于麦卡锡的原始构想，是由拉塞尔自行添加的。它们从此使得Lisp语言看上去很古怪，但也成为了这种语言最独一无二的特点。说Lisp语言古怪倒不是因为它的语法很古怪，而是因为它根本没有语法，程序直接以解析树（parse tree）的形式表达出来。在其他语言中，这种形式只是经过解析在后台产生，但是Lisp直接采用它作为表达形式。它由列表构成，而列表则是Lisp的基本数据结构。



用一门语言自己的数据结构来表达该语言是非常强大的功能。思想(8)和思想(9)，意味着你可以写出一种能够自己编程的程序。这可能听起来很怪异，但是对于Lisp语言却是再普通不过。最常用的做法就是使用宏。

术语“宏”在Lisp语言中的意思与其他语言中的不一样。Lisp宏无所不包，它既可能是某样表达式的缩略形式，也可能是一种新语言的编译器。无论是想真正理解Lisp语言，还是只想拓宽编程视野，最好都学学宏。

就我所知，宏（采用Lisp语言的定义）目前仍然是Lisp独有的。一个原因是为了使用宏，你大概不得不让你的语言看上去像Lisp一样古怪。另一个可能的原因是，如果你想为自己的语言添上这种终极武器，你从此就不能声称自己发明了新语言，只能说发明了一种Lisp的新方言。

---

① 如果你不想让经理发现你正在使用Lisp编程，你可以告诉他你用的是XML。

我把这件事当作笑话说出来，但是事实就是如此。如果你创造了一种新语言，其中有 car、cdr、cons、quote、cond、atom、eq 这样的功能，还有一种把函数写成列表的表示方法，那么在它们的基础上完全可以推导出 Lisp 语言的所有其他部分。事实上，Lisp 语言就是这样定义的，麦卡锡把语言设计成这个样子就是为了让这种推导成为可能。

语言优势真正体现的地方

就算 Lisp 确实代表了目前主流编程语言不断靠近的一个方向，这是否意味着你就应该用它编程呢？如果使用一种不是如此强大的语言，你又会有多少损失呢？有时不采用最尖端的技术不也是一种明智的选择吗？这么多人使用主流编程语言，这本身不也说明那些语言有可取之处吗？举例来说，你的经理不正是希望使用一种很容易雇到程序员的语言吗？



另一方面，许多项目是无所谓选择哪一种编程语言，反正不同的语言都能完成工作。一般来说，条件越苛刻的项目，强大的编程语言就越能发挥作用。但是，无数的项目根本没有苛刻条件的限制。大多数的编程任务可能只要写一些很小“胶水程序”，然后再把这些小程序连起来就行了。你可以用自己熟悉的编程语言或者用对于特定项目来说有着最强大函数库的语言来写这些“胶水程序”。如果你只是需要在 Windows 应用程序之间传递数据，使用 Visual Basic 照样能达到目的。

你也可以使用 Lisp 语言编写这些小程序（我用它写了桌面计算器），但是 Lisp 的最大优势体现在编程任务的另一端，就是在激烈竞争的条件下开发那些解决困难问题的复杂程序。ITA 软件公司为 Orbitz 旅行社开发的飞机票价搜索程序就是一个很好的例子。网络订票市场很难进入，因为它已经被两大巨头（Travelocity 和 Expedia）牢牢控制了，但是 ITA 的软件性能看上去使得那两家公司的软件顿时相形见绌。

ITA 的软件的核心是一个 20 万行的 Common Lisp 程序，它的搜索能力比竞争对手高出许多个数量级。那些竞争对手依然使用大型机时代的编程方法。我没有看过 ITA 的软件源码，但是据一个为它工作的顶尖黑客说，他们使用了大量的宏。果然不出我所料。

向 心 力

我承认，使用一种不常用的技术也有代价。你的经理担心这一点并不是完全没有道理的。但是，因为他不懂风险出在什么地方，所以往往把风险夸大了。

使用一种不常见的语言会出现的问题我想到了三个：你的程序可能无法很好地与使用其他语言写的程序协同工作；你可能找不到很多函数库；你可能不容易雇到程序员。

它们有多严重？第一个问题取决于你是否控制整个系统。如果你的软件运行在客户的机器上，而客户又使用一个到处都是 bug 的专有操作系统（我可没提操作系统的名字），那么使用那个操作系统的开发语言可能会给你带来优势。但是，如果你控制整个系统，并且还有各个组成部分的源码（正如我推测 ITA 就是这种情况），那么你就能使用任何你想用的语言。如果出现不兼容的情况，你自己就能动手解决。



把软件运行在服务器端就可以没有顾忌地使用最先进的技术。乔纳森·埃里克森说现在是“编程语言的文艺复兴时期”，我想最大的原因就是有了服务器端软件。这也能解释为什么像 Perl 和 Python 这样的新语言会流行起来，它们之所以流行不是因为人们使用它们开发 Windows 应用程序，而是因为人们在服务器上使用它们。随着软件从桌面端向服务器端转移（连微软公司都看出这是未来的趋势），逼迫你使用某一种语言的限制将越来越少。

至于第二个问题，函数库的重要性也取决于你的应用程序。对于那些条件不苛刻的应用，有没有一个好的函数库比语言本身的能力更重要。那么到底应该怎么选择语言？是根据函数库，还是根据语言本身的能力？很难确切地找出一条清楚的规则，但是无论哪种情况，你都必须考虑到你开发的应用程序的特点。如果你是一家软件公司，你开发的程序打算拿到市场上销售，那么这个程序可能会耗费好几个优秀程序员至少6个月的时间。为一个这样规模的项目选择编程语言，语言本身要有强大的编程能力可能就是最重要的考虑因素，比是否有方便的函数库更重要。

第三个问题是你的经理担忧雇不到程序员，我认为这根本就是混淆

视听。说实话，你究竟想雇用多少个黑客？到目前为止，大家公认少于10个人的团队最适合开发软件。雇用这样规模的开发团队，只要使用的不是无人知道的语言，应该都不会遇到很大麻烦。如果你无法找到10个Lisp黑客，那么你可能选错了创立软件公司的城市。

事实上，选择更强大的编程语言会减少所需要的开发人员数量。因为：(a) 如果你使用的语言很强大，可能会减少一些编程的工作量，也就不需要那么多黑客了；(b) 使用更高级语言的黑客可能比别的程序员更聪明。

我不是说外界因素对你没有影响，肯定还是会有很大压力，逼迫你使用公认的"标准"技术。Viaweb创业期间，很多风险投资商和潜在的并购方看到我们使用Lisp语言都感到很吃惊和不以为然。但是，我们让他们吃惊的还不止这一个地方，我们使用普通的兼容机充当服务器，而不是"企业级"的Sun服务器；我们使用那时还默默无闻的开源Unix系统FreeBSD，而不是流行的商业操作系统Windows NT；我们也没有采用SET（Secure Electronic Transaction，安全电子交易）,它被认为将成为电子商务标准，而实际上现在没人记得它。诸如此类的事情还有很多。



你不能让那些衣冠楚楚、西装革履的家伙替你做技术决策。潜在的并购方有没有对我们使用Lisp语言感到很难接受？稍微有一点吧，但是如果我们不使用Lisp，我们就根本写不出现在的软件，也就不会有人想收购我们。他们眼中不正常的事情恰恰就是使得这一切发生的原因所在。

如果你创业的话，千万不要为了取悦风险投资商或潜在并购方而设计你的产品。让用户感到满意才是你的设计方向。只要赢得用户，其他事情就会接踵而来。如果没有用户，谁会关心你选择的"正统"技术是多么令人放心。

随大流的代价

使用一种不强大的语言，你的损失有多大？实际上有一些现成的数据可以说明这个问题。

衡量语言的编程能力的最简单方法可能就是看代码数量。所谓高级语言，就是能够提供更强大抽象能力的语言，从某种意义上，就像能够

提供更大的砖头，所以砌墙的时候用到的砖头数量就变少了。因此，语言的编程能力越强大，写出来的程序就越短（当然不是指字符数量，而是指独立的语法单位）。

强大的编程语言如何让你写出更短的程序？一个技巧就是（在语言允许的前提下）使用“自下而上”（bottom-up）的编程方法。你不是用基础语言（base language）开发应用程序，而是在基础语言之上先构建一种你自己的语言，然后再用后者开发应用程序。这样写出来的代码会比直接用基础语言开发出来的短得多。实际上，大多数压缩算法也是这样运作的。“自下而上”的编程往往也便于修改，因为许多时候你自己添加的中间层根本不需要变化，你只需要修改前端逻辑就可以了。

代码的数量很重要，因为开发一个程序所耗费的时间主要取决于程序的长度。对于同一个软件，如果用一种语言写出来的代码比用另一种语言长三倍，这意味着你开发它耗费的时间也会多三倍。而且即使多雇人手，也无助于缩短开发时间，因为当团队规模超过某个门槛时，再增加人手只会带来净损失。Fred Brooks 在他的名著《人月神话》中描述了这种现象，我的所见所闻印证了他的说法。



如果使用 Lisp 语言，程序能变得多短？以 Lisp 和 C 的比较为例，我听到的大多数说法是 C 代码的长度是 Lisp 的 7 倍到 10 倍。但是最近，*New Architect* 杂志上有一篇介绍 ITA 软件公司的文章[①]，里面说“1 行 Lisp 代码相当于 20 行 C 代码”，因为此文都是引用 ITA 总裁的话，所以我想这个数字来自 ITA 的编程实践。如果真是这样，那么我们可以相信这句话。ITA 的软件不仅使用 Lisp 语言，还同时大量使用 C 和 C++，所以这是他们的经验之谈。

我认为，这种比例肯定不会是一个常数。如果你遇到更困难的问题，或者你雇到了更聪明的程序员，这个比例就会增大。一种出色的工具到了真正优秀的黑客手里，可以发挥出更大的威力。

总之，根据上面的这个数字，如果你与 ITA 竞争，而且你使用 C 语言开发软件，那么 ITA 的开发速度将比你快 20 倍。如果你需要一年时间

---

① Jen Muehlbauer，“Orbitz的新突破”（*Orbitz Reaches New Heights*），*New Architect*，2002年4月号。

实现某个功能，它只需要不到三星期。反过来说，如果 ITA 开发某个新功能用了三个月，那么你需要五年才能做出来。

你知道吗？上面的对比还只是考虑到最好的情况。当我们只比较代码数量的时候，言下之意就是假设使用功能较弱的语言也能开发出同样的软件。但是事实上，程序员使用某种语言能做到的事情是有极限的。如果你想用一种低层次的语言解决一个很难的问题，那么你将会面临各种情况极其复杂乃至想不清楚的窘境。

所以，当我说假定你与 ITA 竞争，你用五年时间做出的东西，ITA 在 Lisp 语言的帮助下只用三个月就完成了，我指的五年还是一切顺利、没有犯错误、也没有遇到太大麻烦的五年。事实上，按照大多数公司的实际情况，计划中五年完成的项目很可能永远都不会完成。

我承认，上面的例子太极端。ITA 似乎有一批非常聪明的黑客，而 C 语言又是一种很低层次的语言。但是，在一个高度竞争的市场中，即使开发速度只相差两三倍，也足以使得你永远处在落后的位置。



一个诀窍

由于选择了不当的编程语言而导致项目失败的可能性，是你的经理不愿意考虑的问题。事实上大部分的经理都这样。因为你知道，总的来说，你的经理其实不关心公司是否真的能获得成功，他真正关心的是不承担决策失败的责任。所以对他个人来说，最安全的做法就是跟随大多数人的选择。

在大型组织内部，有一个专门的术语描述这种跟随大多数人的选择的做法，叫做“业界最佳实践”。这个词出现的原因其实就是为了让你的经理可以推卸责任。既然我选择的是“业界最佳实践”，如果不成功，项目失败了，那么你也无法指责我，因为做出选择的人不是我，而是整个“业界”。

我认为这个词原来是指某种会计方法，大致意思就是不要采用很奇怪的处理方法。在会计方法中，这可能是一个很好的主意。“尖端”和“核算”这两个词听上去就不适合放在一起。但是如果你把这个标准引入技术决策，你就开始要出错了。

技术本来就应该是尖端的。正如伊拉恩·加内特所说，编程语言的所谓“业界最佳实践”，实际上不会让你变成最佳，只会让你变得很平常。如果你选择的编程语言使得你开发软件的速度只有(选择更激进技术的)对手的几分之一，那么“最佳实践”真的起错了名字。

所以，我们就有了两点结论，我认为它们非常有价值。事实上，这是我用自己的经历换来的。第一，不同语言的编程能力不一样。第二，大多数经理故意忽视第一点。你把这两点事实结合起来，其实就得到了赚钱的诀窍。ITA 软件公司是运用这个诀窍的典型例子。如果你想在软件业获得成功，就使用你知道的最强大的语言，用它解决你知道的最难的问题，并且等待竞争对手的经理做出自甘平庸的选择。

附录：编程能力

为了解释我所说的语言编程能力不一样，请考虑下面的问题。我们需要写一个函数，它能够生成累加器，即这个函数接受一个参数 n，然后返回另一个函数，后者接受参数 i，然后返回 n 增加（increment）了 i 后的值。[这里说的是增加，而不是 n 和 i 的相加（plus）。累加器就是应该完成 n 的累加。]



Common Lisp[①]的写法如下：

```
(defun foo (n)
  (lambda (i) (incf n i)))
```

Ruby 的写法几乎完全相同：

```
def foo (n)
  lambda {|i| n += i } end
```

Perl 5 的写法则是：

```
sub foo {
  my ($n) = @_;
  sub {$n += shift}
}
```

① 下面是一些Lisp方言生成累加器函数的写法：

```
Scheme: (define (foo n)
          (lambda (i) (set! n (+ n i)) n))
Goo:    (df foo (n) (op incf n _)))
Arc:    (def foo (n) [++ n _])
```

这比 Lisp 和 Ruby 的版本有更多的语法元素，因为在 Perl 语言中必须手工提取参数。

Smalltalk 的写法比 Lisp 和 Ruby 的稍微长一点：

```
foo: n
  |s|
  s := n.
  ^[:i| s := s+i. ]
```

因为在 Smalltalk 中，词法变量（lexical variable）[①]是有效的，但是你无法给一个参数赋值，因此不得不设置了一个新变量，接受累加后的值。

JavaScript 的写法也比 Lisp 和 Ruby 稍微长一点，因为 JavaScript 依然区分语句和表达式，所以需要明确指定 `return` 语句来返回一个值：

```
function foo (n) {
  return function (i) {
           return n += i } }
```

（实事求是地说，Perl 也保留了语句和表达式的区别，但是使用了常规的 Perl 方式处理，因此可以省略 `return`。）



如果想把 Lisp/Ruby/Perl/Smalltalk/JavaScript 的版本改成 Python，你会遇到一些限制。因为 Python 并不完全支持词法变量，你不得不创造一种数据结构来接受 *n* 的值。而且尽管 Python 确实支持函数数据类型，但是没有一种字面量的表示方式（literal representation）可以生成函数（除非函数体只有一个表达式），所以你需要创造一个命名函数，把它返回。最后的写法如下：

```
def foo (n):
  s = [n]
  def bar (i):
    s[0] += i
    return s[0]
  return bar
```

Python 用户完全可以合理地质疑为什么不能写成下面这样：

```
def foo (n):
  return lambda i: return n += i
```

或者

```
def foo (n):
  lambda i: n += i
```

① 词法变量，指的是变量的作用域由代码结构决定，不取决于运行时的调用顺序。也就是说，作用域在代码文本的词法分析阶段就决定了，而不在代码执行时决定。注意将这个概念与“局部变量”的概念相区分。——译者注

我猜想，Python 有一天会支持这样的写法。（如果不想等到 Python 慢慢进化到更像 Lisp，总可以直接……）

在面向对象编程的语言中，你能够在有限程度上模拟一个闭包（即一个函数，通过它可以引用由包含这个函数的代码所定义的变量）。你定义一个类（class），里面有一个方法和一个属性，用于替换封闭作用域（enclosing scope）中的所有变量。这有点类似于让程序员自己做代码分析，本来这应该是由支持词法作用域（lexical scope）的编译器完成的。如果有多个函数，同时指向相同的变量，那么这种方法就会失效，但是在这个简单的例子中，它已经足够了。

Python高手看来也同意这是解决这个问题比较好的方法，写法如下：

```
def foo (n):
  class acc:
    def __init__ (self, s):
        self.s = s
    def inc (self, i):
        self.s += i
        return self.s
  return acc (n).inc
```



或者

```
class foo:
  def __init__ (self, n):
      self.n = n
  def __call__ (self, i):
      self.n += i
      return self.n
```

我添加这一段是想避免 Python 爱好者说我误解这种语言。但是在我看来，这两种写法好像都比第一个版本更复杂。你实际上就是在做同样的事，只不过划出了一个独立的区域保存累加器函数，区别只是保存在对象的一个属性中，而不是保存在列表（list）的头（head）中。使用这些特殊的内部属性名（尤其是`__call__`）看上去并不像常规的解法，更像是一种破解。

在Perl和Python的较量中，Python黑客的观点似乎是认为Python比Perl更优雅，但是这个例子表明，最终来说，编程能力决定了优雅程度。Perl的写法更简单（包含的语法元素更少），尽管它的语法有一点丑陋。

其他语言怎么样？前文曾经提到过 Fortran、C、C++、Java 和 Visual Basic，看上去使用它们根本无法解决这个问题。肯·安德森说，Java 只能写出一个近似的解法：

```
public interface Inttoint {
  public int call (int i);
}

public static Inttoint foo (final int n) {
  return new Inttoint () {
    int s = n;
    public int call (int i) {
    s = s + i;
    return s;
    }};
}
```

这种写法不符合题目要求，因为它只对整数有效。

当然，我说使用其他语言无法解决这个问题，这句话并不完全正确。所有这些语言都是图灵等价的，这意味着严格地说，你能使用它们之中的任何一种语言写出任何一个程序。那么，怎样才能做到这一点呢？就这个小小的例子而言，你可以使用这些不那么强大的语言写一个 Lisp 解释器就行了。



这样做听上去好像开玩笑，但是在大型编程项目中却不同程度地广泛存在。因此，有人把它总结出来，起名为“格林斯潘第十定律”（Greenspun’s Tenth Rule）：

> 任何C或Fortran程序复杂到一定程度之后，都会包含一个临时开发的、只有一半功能的、不完全符合规格的、到处都是bug的、运行速度很慢的Common Lisp实现。

如果你想解决一个困难的问题，关键不是你使用的语言是否强大，而是好几个因素同时发挥作用：（a）使用一种强大的语言；（b）为这个难题写一个事实上的解释器；或者（c）你自己变成这个难题的人肉编译器。在 Python 的例子中，这样的处理方法已经开始出现了，我们实际上就是自己写代码，模拟出编译器实现词法变量的功能。

这种实践不仅很普遍，而且已经制度化了。举例来说，在面向对象编程的世界中，我们大量听到“模式”（pattern）这个词，我觉得那些“模

式”就是现实中的因素（c），也就是人肉编译器。[①]当我在自己的程序中发现用到了模式，我觉得这就表明某个地方出错了。程序的形式应该仅仅反映它所要解决的问题。代码中其他任何外加的形式都是一个信号，（至少对我来说）表明我对问题的抽象还不够深，也经常提醒我，自己正在手工完成的事情，本应该写代码通过宏的扩展自动实现。



① 皮特·诺维格发现，总共23种设计模式之中，有16种在Lisp语言中“本身就提供，或者被大大简化”。（www.norvig.com/design-patterns）

14

梦寐以求的编程语言

一心让臣民行善的暴君可能是最专制的暴君。

——C. S. LEWIS（1898—1963，英国小说家）



我的朋友曾对一位著名的操作系统专家说他想要设计一种真正优秀的编程语言。那位专家回答，这是浪费时间，优秀的语言不一定会被市场接受，很可能无人使用，因为语言的流行不取决于它本身。至少，那位专家设计的语言就遭遇到了这种情况。

那么，语言的流行到底取决于什么因素呢？流行的语言是否真的值得流行呢？还有必要尝试设计一种更好的语言吗？如果有必要的话，怎样才能做到这一点呢？

为了找到这些问题的答案，我想我们可以观察黑客，了解他们使用什么语言。编程语言本来就是为了满足黑客的需要而产生的，当且仅当黑客喜欢一种语言时，这种语言才能成为合格的编程语言，而不是被当作“指称语义”（denotational semantics）或者编译器设计。

流行的秘诀

没错，大多数人选择某一种编程语言，不是因为这种语言有什么独特的特点，而是因为听说其他人使用这种语言。但是我认为，外界因素对于编程语言的流行其实没有想象中那么大的影响力。我倒是觉得，问题出在对于什么是优秀编程语言，黑客的看法与大多数的语言设计者不一样。

黑客的看法其实比语言设计者的更重要。编程语言不是数学定理，而是一种工具，为了便于使用，它们才被设计出来。所以，设计编程语言的时候必须考虑到人类的长处和短处，就像设计鞋子的时候必须符合人类的脚型。如果鞋子穿上去不舒服，无论它的外形多么优美，多么像一件艺术品，你也只能把它当作一双坏鞋。

大多数程序员也许无法分辨语言的好坏。但是，这不代表优秀的编程语言会被埋没，专家级黑客一眼就能认出它们，并且会拿来使用。虽然他们人数很少，但就是这样一小群人写出了人类所有优秀软件。他们有着巨大的影响力，他们使用什么语言，其他程序员往往就会跟着使用。老实说，很多时候这种影响力更像是一种命令，对于其他程序员来说，专家级黑客就像自己的老板或导师，他们说哪种语言好用，自己就会乖乖地跟进。

专家级黑客的看法不是决定一种语言流行程度的唯一因素，某些古老的软件（Fortran 和 Cobol 的情况）和铺天盖地的广告宣传（Ada 和 Java 的情况）也会起到作用。但是，我认为从长期来看，专家级黑客的看法是最重要的因素。只要有了达到“临界数量”（critical mass）的最初用户和足够长的时间，一种语言可能就会达到应有的流行程度。而流行本身又会使得这种优秀的语言更加优秀，进一步拉大它与平庸语言之间的好坏差异，因为使用者的反馈总是会导致语言的改进。你可以想一下，所有流行的编程语言从诞生至今的变化有多大。Perl 和 Fortran 是极端的例子，除它们两个之外，甚至就连 Lisp 都发生了很大的变化。



所以，即使不考虑语言本身的优秀是否能带动流行，我想单单流行本身就肯定会使得这种语言变得更好，只有流行才会让它保持优秀。编程语言的最高境界一直在发展之中。虽然语言的核心功能就像大海的深处，很少有变化，但是函数库和开发环境之类的东西就像大海的表面，一直在汹涌澎湃。

当然，黑客必须先知道这种语言，才可能去用它。他们怎么才能知道呢？就是从其他黑客那里。所以不管怎样，一开始必须有一群黑客使用这种语言，然后其他人才会知道它。我不知道“一群”的最小数量是多少，多少个黑客才算达到“临界数量”呢？如果让我猜，我会说 20

人。如果一种语言有 20 个独立用户，就意味这 20 个人是自主决定使用这种语言的，我觉得这就说明这种语言真的有优点。

达到这一步并非易事。如果说用户数从 0 到 20 比从 20 到 1000 更困难，我也不会感到惊讶。发展最早的 20 个用户的最好方法可能就是使用特洛伊木马：你让人们使用一种他们需要的应用程序，这个程序偏巧就是用某种新语言开发的。

外部因素

我们得先承认，确实有一个外部因素会影响到语言的流行。一种语言必须是某一个流行的计算机系统的脚本语言（scripting language），才会变得流行。Fortran和Cobol是早期IBM大型机的脚本语言。C是Unix的脚本语言，后来的Perl和Python也是如此。Tcl是Tk的脚本语言，Visual Basic是Windows的脚本语言，（某种形式的）Lisp是Emacs的脚本语言，PHP是网络服务器的脚本语言，Java和JavaScript是浏览器的脚本语言。



编程语言不是存在于真空之中。“编程”其实是及物动词，黑客一般都是为某个系统编程，在现实中，编程语言总是与它们依附的系统联系在一起的。所以，如果你想设计一种流行的编程语言，就不能只是单纯地设计语言本身，还必须为它找到一个依附的系统，而这个系统也必须流行。除非你只想用自己设计的语言取代那个系统现有的脚本语言。

这种情况导致的一个结果就是，无法以一种语言本身的优缺点评判这种语言。另一个结果则是，只有当一种语言是某个系统的脚本语言时，它才能真正成为编程语言。如果你对此很吃惊，觉得不公平，那么我会跟你说不必大惊小怪。这就好比大家都认为，如果一种编程语言只有语法规则，没有一个好的实现（implementation），那么它就不能算完整的编程语言。这些都是很正常很合理的事情，编程语言本来就该如此。

当然，编程语言本来就需要一个好的实现，而且这个实现必须是免费的。商业公司愿意出钱购买软件，但是黑客作为个人不会愿意这样做，而你想让一种语言成功，恰恰就是需要吸引黑客。

编程语言还需要有一本介绍它的书。这本书应该不厚，文笔流畅，而且包含大量优秀的范例。布赖恩·柯尼汉和丹尼斯·里奇合写的《C

程序设计语言》（*C Programming Language*）就是这方面的典范。眼下，我大概还能再加一句，这一类书籍之中必须有一本由 O'Reilly 公司出版发行。这正在变成是否能吸引黑客的前提条件了。

编程语言还应该有在线文档。事实上，在线文档可以当作一本书来写，但是目前它还无法取代实体书。实体书并没有过时，它们读起来很方便，而且出版社对书籍内容的审核是一种很有用的质量保证机制（虽然做得很不完美）。书店则是程序员发现和学习新语言的最重要的场所之一。

简　　洁

假定你的语言已经能够满足上面三项条件——一种免费的实现，一本相关书籍，以及语言所依附的计算机系统——那么还需要做什么才能使得黑客喜欢上你的语言？

黑客欣赏的一个特点就是简洁。黑客都是懒人，他们同数学家和现代主义建筑师一样，痛恨任何冗余的东西或事情。有一个笑话说，黑客动手写程序之前，至少会在心里盘算一下哪种语言的打字工作量最小，然后就选择使用该语言。这个笑话其实与真实情况相差无几。就算这真的是个笑话，语言的设计者也必须把它当真，按照它的要求设计语言。

简洁性最重要的方面就是要使得语言更抽象。为了达到这一点，首先你设计的必须是高级语言，然后把它设计得越抽象越好。语言设计者应该总是看着代码，问自己能不能使用更少的语法单位把它表达出来。如果你有办法让许多不同的程序都能更简短地表达出来，那么这很可能意味着你发现了一种很有用的新抽象方法。

不要觉得为用户着想就是让他们使用像英语一样又长又啰嗦的语法。这是不正确的做法，Cobol 就是因为这个毛病而声名狼藉。如果你让黑客像下面这样求和：

```
add x to y giving z
```

而不是写成：

```
z=x+y
```

那么你就是在侮辱黑客的智商，或者自己作孽了。



简洁性是静态类型语言的力所不及之处。不考虑其他因素时，没人愿意在程序的头部写上一大堆的声明语句。只要计算机可以自己推断出来的事情，都应该让计算机自己去推断。举例来说，hello-world 本应该是一个很简单的程序，但是在 Java 语言中却要写上一大堆东西，这本身就差不多可以说明 Java 语言设计得有问题了。[①]

单个的语法单位也应该很简短。Perl 和 Common Lisp 在这方面是两个不同的极端。Perl 的语法单位很短，导致它的代码可以拥挤得让人无法理解，而 Common Lisp 内置运算符的名称则长得可笑。Common Lisp 的设计者们可能觉得文本编辑器会帮助用户自动填写运算符的长名称。但是这样做的代价不仅是增加了打字的工作量，还包括提高了阅读代码的难度，以及占用了更多的显示器空间。

可编程性（Hackability）



对黑客来说，选择编程语言的时候，还有一个因素比简洁更重要，那就是这种语言必须能够帮助自己做到想做的事。在编程语言的历史上，防止程序员做出“错误”举动的措施多得惊人。这是语言设计者很自以为是的危险举动，他们怎么知道程序员该做什么不该做什么？我认为，语言设计者应该假定他们的目标用户是一个天才，会做出各种他们无法预知的举动，而不是假定目标用户是一个笨手笨脚的傻瓜，需要别人的保护才不会伤到自己。如果用户真的是傻瓜，不管你怎么保护他，他还是会搬起石头砸自己的脚。你也许能够阻止他引用另一个模块中的变量，但是你没法防止他日日夜夜不知疲倦地写出结构混乱的程序去解决完全错误的问题。

---

① hello-world程序的唯一作用就是显示出“Hello, world!”这句话。使用Java语言，你需要这样写：

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

如果你从来没有接触过编程，看到上面的代码可能会很奇怪，让计算机显示一句话为什么要搞得这么复杂？有意思的是，资深程序员的反应与你一样。

优秀程序员经常想做一些既危险又令人恼火的事情。所谓“令人恼火”，我指的是他们会突破设计者提供给用户的外部语义层，试着控制某些高级抽象的语言内部接口。比如，黑客喜欢破解，而破解就意味着深入内部，揣测原始设计者的意图。

你应该敞开胸怀，欢迎这种揣测。对于制造工具的人来说，总是会有用户以违背你本意的方式使用你的工具。如果你制造的是编程语言这样高度组合的系统，那就更是如此了。许多黑客会用你做梦也想不到的方式改动你的语法模型。我的建议就是，让他们这样干吧，而且应该为他们创造便利，尽可能多地把语言的内部暴露在他们面前。

其实，黑客并不会彻底颠覆你的工具，在一个大型程序中，他可能只是对语言改造一两个地方。但是，改动多少地方并不重要，重要的是他能够对语言进行改动。这可能不仅有助于解决一些特殊的问题，还会让黑客觉得很好玩。黑客改造语言的乐趣就好比外科医生摆弄病人内脏的乐趣，或者青少年喜欢用手挤破青春痘的那种感觉。①至少对男生来说，某些类型的破坏非常刺激。针对青年男性读者的 *Maxim* 杂志每年出版一本特辑，里面一半是美女照片，另一半是各种严重事故的现场照片。这本杂志非常清楚它的读者想看什么。

一种真正优秀的编程语言应该既整洁又混乱。“整洁”的意思是设计得很清楚，内核由数量不多的运算符构成，这些运算符易于理解，每一个都有很完整的独立用途。“混乱”的意思是它允许黑客以自己的方式使用。C 语言就是这样的例子，早期的 Lisp 语言也是如此。真正的黑客语言总是稍微带一点放纵不羁、不服管教的个性。



---

① 在《神经外科医生手记》（*When the Air Hits Your Brain*）一书中，神经外科医生弗托塞克讲述了住院总医生戈雷的一段话，内容关于外科医生与内科医生的区别。

> 戈雷和我要了一个大比萨，找了一张空桌子坐下。他点起一根香烟，说：“那些内科医生真是令人讨厌，总是喜欢谈论一辈子只能遇到一次的病例。这就是他们的问题，他们只喜欢古怪的东西，讨厌普通的常见病例。这就是我们和他们的区别。你看，我们喜欢腰椎间盘突出，觉得像比萨一样又大又好吃，但是他们看到高血压就憎恨不已……”

很难把腰椎间盘突出与又大又好吃联系在一起，但是，我想我知道他们指的是什么。我经常觉得某个bug非常诱人，一定要追踪下去。不是程序员的人很难想象bug有什么好玩的。一切正常当然很好，但是不可否认，能够抓到某些bug会让人兴奋到极点。

优秀的编程语言所具备的功能，应该会使得言必称“软件工程”的人感到非常不满、频频摇头。与黑客语言形成鲜明对照的就是像 Pascal 那样的语言，它是井然有序的模范，非常适合教学，但是除此之外就没有很大用处了。

一次性程序

为了吸引黑客，一种编程语言必须善于完成黑客想要完成的各种任务。这意味着它必须很适合开发一次性程序。这一点可能出乎很多人的意料。

所谓一次性程序，就是指为了完成某些很简单的临时性任务而在很短时间内写出来的程序。比如，自动完成某些系统管理任务的程序，或者（为了某项模拟任务）自动生成测试数据的程序，以及在不同格式之间转化数据的程序等。令人吃惊的是，一次性程序往往不是真的只用一次，就像二战期间很多美国大学造的一大批临时建筑后来都成了永久建筑。许多一次性程序后来也都变成了正式的程序，具备了正式的功能和外部用户。



我有一种预感，最优秀的那些大型程序就是这样发展起来的，而不是像胡佛水坝那样从一开始就作为大型工程来设计。一下子从无到有做出一个大项目是很恐怖的一件事。当人们接手一个巨型项目时，很容易被它搞得一蹶不振。最后，要么是项目陷入僵局，要么是做出来一个规模小、性能差的东西。你想造一片闹市，却只做出一家商场；你想建一个罗马，却只造出一个巴西利亚；你想发明 C 语言，却只开发出 Ada。

开发大型程序的另一个方法就是从一次性程序开始，然后不断地改进。这种方法比较不会让人望而生畏，程序在不断的开发之中逐渐进步。一般来说，使用这种方法开发程序，一开始用什么编程语言，就会一直用到最后，因为除非有外部政治因素的干预，程序员很少会中途更换编程语言。所以，我们就有了一个看似矛盾的结论：如果你想设计一种适合开发大型项目的编程语言，就必须使得这种语言也适合开发一次性程序，因为大型项目就是从一次性程序演变而来的。

Perl 就是一个鲜明的例子。它不仅仅设计成适合开发一次性程序，

而且它本身就很像一次性程序。最初的 Perl 只是好几个生成表格的工具收集在一起而已。后来程序员用它写一次性程序，当那些程序逐渐发展壮大后，Perl 才随之发展成了一种正式的编程语言。到了 Perl 5，这种语言才适合开发重要的程序，但是在此之前它已经广为流行了。

什么样的语言适合写一次性程序？首先，它必须很容易装备。一次性程序是你只想在一小时内写出来的程序，所以它不应该耗费很多时间安装和配置，最好已经安装在你的电脑上了。它必须是想用就用的。C 语言可以想用就用，因为它是操作系统的一部分；Perl可以想用就用，因为它本来就是一种系统管理工具，操作系统已经默认安装它了。

很容易装备不仅仅指很容易安装或者已经安装，还指很容易与使用者互动。一种有命令行界面、可以实时反馈的语言就具有互动性，那些必须先编译后使用的语言就不具备互动性。受欢迎的编程语言应该是前者，具有良好的互动性，可以快速得到运行结果。



一次性程序的另一个特点就是简洁。对黑客来说，这一点永远有吸引力。如果考虑到你最多只打算在这个程序上耗费一个小时，这一点就更重要了。

函　数　库

简洁性的最高形式当然是有人已经帮你把程序写好，你只要运行就可以了。函数库就是别人帮你写好的程序，所以它是编程语言的另一个重要特点，并且我认为正在变得越来越重要。Perl 就赢在它具有操作字符串的巨大函数库。这类函数库对一次性程序特别重要，因为开发一次性程序的原始目的往往就是转化或提取字符串。许多 Perl 程序的原型可能就是把几个函数库调用放在一起。

我认为，未来 50 年中，编程语言的进步很大一部分与函数库有关。未来的函数库将像语言内核一样精心设计。优秀函数库的重要性将超过语言本身。某种语言到底是静态类型还是动态类型、是面向对象还是函数式编程，这些都不如函数库重要。那些习惯用变量类型考虑问题的语言设计者可能会对这种趋势感到不寒而栗。这不等于把语言设计降到开发应用程序的层次吗？哦，真是太糟了。但是别忘了，编程语言是供程

序员使用的，而函数库就是程序员需要的东西。

设计优秀的函数库是很难的，并不只是写一大堆代码而已。一旦函数库数量变得太多，找到一个你需要的函数有时候还不如自己动手写来得快。函数库的设计基础与语言内核一样，都是一个小规模的正交运算符集合。函数库的使用应该符合程序员的直觉，让他可以猜得出哪个函数能满足自己的需要。

效　　率

众所周知，好的编程语言生成的代码有较快的运行速度。但是实际上，我觉得代码的运行速度不是编程语言的设计者能够控制的。高德纳很久以前就指出，运行速度只取决于一些关键的瓶颈。而在编程实践中，许多程序员都已经注意到自己很容易搞错瓶颈到底在哪里。



所以，编程时提高代码运行速度的关键是使用好的性能分析器（profiler），而不是使用其他方法，比如精心选择一种静态类型的编程语言。为了提高运行速度，并没有必要每个函数的每个参数类型都声明清楚，你只需要在瓶颈处声明清楚参数类型就可以了。所以，更重要的是你需要能够找出瓶颈到底在什么地方。

人们在使用非常高级的语言（比如 Lisp）时，经常抱怨很难知道哪个部分对性能的影响比较大。可能确实如此，如果你使用一种非常抽象的语言，这也许是无法避免的。不管怎样，我认为一个好的性能分析器会解决这个问题，虽然这方面还有很长的路要走，但是未来你可以快速知道程序每个部分的时间开销。

这个问题一部分源于沟通不畅。语言设计者喜欢提高编译器的速度，认为这是对自己技术水平的考验，而最多只把性能分析器当作一个附送给使用者的赠品。但是在现实中，一个好的性能分析器对程序的帮助可能大于编译器的作用。这里又一次反映出语言设计者与用户之间发生了脱节，前者竭尽全力想要解决的问题其实方向不甚正确。

让性能分析器自动运行可能是一个好主意。它自动告诉程序员每个部分的性能，而不是非要等到程序员手动运行后才能知道。比如，当程序员编辑源码的时候，代码编辑器能够实时用红色显示瓶颈的部分。另

一个方法应该是设法显示正在运行的程序的情况，这对互联网软件尤其重要，因为服务器上有很多程序同时运行，它们都需要你密切关注。自动运行的性能分析器用图形实时显示程序运行时的内存状况，甚至可以发出声音，表示出现了问题。

出现问题时，声音是很好的提示。我们在 Viaweb 搞了一块很大的面板，上面有各种各样的仪表盘，用来显示服务器的状况。仪表盘的指针由微型马达驱动，每当马达旋转的时候，就会发出一阵轻微的噪音。在我的工位没法看到仪表盘，但是只要我听到声音，就能立刻知道服务器出现了问题。

性能分析器甚至有可能自动找出不合理的算法。如果将来有人发现某种形式的内存访问是不合理算法的信号，我不会感到很惊讶。如果有一个小人儿可以钻进计算机看看我们的程序是怎么运行的，他可能会变成一个忙碌又悲惨的可怜虫，就像那些为政府跑腿的小人物。我总觉得自己用处理器做了很多无用功，但是一直没有找到能够看出程序是怎样浪费运算能力的好办法。



现在有一些语言先编译成字节码（byte code），然后再由解释器执行。这样做主要是为了让代码容易移植到不同的操作系统，但是这也可以变成一项很有用的功能。让字节码成为语言的正式组成部分，允许程序员在瓶颈处内嵌字节码，这可能是一个不错的主意。然后，针对这部分字节码的优化也就变得可以移植了。

正如许多最终用户已经意识到的，运行速度的概念正在发生变化。随着互联网软件的兴起，越来越多的程序主要不是受限于计算机的运算速度，而是受限于 I/O 的速度。加快 I/O 速度将是很值得做的一件事。在这方面，编程语言也能起到作用，有些措施是显而易见的，比如采用简洁、快速、格式化输出的函数，还有些措施则需要深层次的结构变化，比如采用缓存和持久化对象（persistent object）。

用户关心的是反应时间（response time），但是软件的另一种效率正在变得越来越重要，那就是每个处理器能够同时支持的用户数量。未来许多有趣的应用程序都将是运行在服务器端的互联网软件，所以每台服务器能够支持的用户数量就成了软件业者的关键问题。互联网软件的成

本支出就取决于这个指标。

许多年以来，大多数面向最终用户的程序都不太关心效率。软件开发者总是假设用户桌面电脑的运算能力会不断增长，所以不用刻意提高软件的效率。帕金森定律[①]被证明与摩尔定律一样颠扑不破。软件不断膨胀，消耗光所有可以得到的资源。这一切将随着互联网软件的出现发生改变，因为硬件和软件现在捆绑在一起供应。对于那些提供互联网软件的公司来说，将每台服务器支持的用户数量最大化会对降低成本产生巨大影响。

在一些应用程序中，处理器的运算能力是瓶颈，那么最重要的优化对象就是软件的运行速度。但是，一般情况下内存才是瓶颈，你能够同时支持的用户数量取决于用户数据所消耗的内存。编程语言在这方面也能发挥作用，对线程的良好支持将使得所有用户共享同一个内存堆（heap）。持久化对象和语言内核级别的延迟加载（lazy loading）支持也有助于减少内存需求。



时　　间

一种编程语言要想变得流行，最后一关就是要经受住时间的考验。没人想用一种会被淘汰的语言编程，这方面已经有很多前车之鉴了。所以，大多数黑客往往会等上几年，看看某一种新语言的势头，然后才真正考虑使用它。

新事物的发明者通常对这个发现很震惊，他们没想到人们居然这样对待发明创造。但是，让别人相信一种新事物是需要时间的。我有一个朋友，他的客户第一次提出某种需求时，他很少理会。因为他知道人们有时候会想要自己并不真正需要的东西。为了避免浪费时间，只有当客户第三次或第四次提出同样的需求时，他才认真对待。这个时候客户可能已经很不高兴了，但是这至少保证他们提出的需求应该就是他们真正需要的东西。

大多数人接触新事物时都学会了使用类似的过滤机制。甚至有时要

① 帕金森定律（Parkinson’s Law）的一种原始表达形式是“工作总是到最后一刻才会完成”，后来引申到计算机领域就变成了“数据总是会填满所有空间”，更一般性的总结则是“对一种资源的需求总是会消耗光这种资源的所有供应”。——译者注

听到别人提起十遍以上他们才会留意。这样做完全是合理的，因为大多数的热门新商品事后被证明都是浪费时间的噱头，没多久就消失得无影无踪。虚拟现实建模语言 VRML 刚诞生时曾经轰动一时，但是我决定等到一两年后再去学习它，结果一两年后已经没有学习的必要了，因为市场已经把它遗忘了。

所以，发明新事物的人必须有耐心，要常年累月不断地做市场推广，直到人们开始接受这种发明。我们就耗费了好几年才使得客户明白 Viaweb 不需要下载安装就能使用。不过，好消息是，简单重复同一个信息就能解决这个问题。你只需要不停地重复同一句话，最终人们将会开始倾听。人们真正注意到你的时候，不是第一眼看到你站在那里，而是发现过了这么久你居然还在那里。

新事物的发展改进一般也需要很长时间。大多数技术在诞生后都逐渐发生了巨大的变化，编程语言更是如此。诞生头几年，一小批早期使用者比其他因素更能促进技术发展。早期使用者都是行家，要求也很高，能够很快找出你的技术中存在的缺点。而且，如果你的用户只有很少几个人，你就能够与他们所有人保持密切接触。只要不断改进你的系统，即使给用户造成了损失，早期使用者也会对你宽容大度的。



新技术被市场接纳的方式有两种，一种是自然成长式，另一种是大爆炸式。自然成长式的一个例子就是在车库里白手起家、自力更生的创业者。几个好朋友埋头工作，在外界毫不知晓的情况下开发出某种新技术。他们把它推向市场，没有任何宣传，最初的用户寥寥无几（但是热心程度无与伦比）。创业者持续改进新技术，与此同时，通过口碑效应，用户数量不断增长。在创业者不经意间，他们已经壮大起来了。

大爆炸式的例子是有风险资本支持、在市场上大张旗鼓宣传的创业公司。他们急急忙忙地开发一个产品，推向市场的时候大肆曝光，立刻就获得了一大批使用者（至少他们希望如此）。

一般来说，车库里的创业者会妒忌大爆炸式的创业公司。后者的主导人物个个光彩照人、自信非凡，深受风险资本商的追捧。他们什么都买得起，在公关公司配合产品推出的宣传活动中，他们自己也附带成为了明星人物。自然成长式的创业者坐在自家车库里，觉得自己又穷又可

怜。但是我想他们不必难过。最终来看，自然成长式会比大爆炸式产生更好的技术，能为创始人带来更多的财富。如果你研究一下目前的主流技术，就会发现大部分都是源于自然成长式。

这种模式不仅存在于商业公司，还存在于科研活动中。Multics 操作系统和 Ada 语言是大爆炸式项目，现在都已经销声匿迹了，而它们的继承者 Unix 和 C 语言则是自然成长式项目。

再 设 计

著名散文家 E.B.怀特说过，"最好的文字来自不停的修改"。所有优秀作家都知道这一点，它对软件开发也适用。设计一样东西，最重要的一点就是要经常"再设计"，编程尤其如此，再多的修改都不过分。

为了写出优秀软件，你必须同时具备两种互相冲突的信念。一方面，你要像初生牛犊一样，对自己的能力信心万丈；另一方面，你又要像历经沧桑的老人一样，对自己的能力抱着怀疑态度。在你的大脑中，有一个声音说"千难万险只等闲"，还有一个声音却说"早岁哪知世事艰"。



这里的难点在于你要意识到，实际上这两种信念并不矛盾。你的乐观主义和怀疑倾向分别针对两个不同的对象。你必须对解决难题的可能性保持乐观，同时对当前解法的合理性保持怀疑。

做出优秀成果的人，在做的过程中常常觉得自己做得不够好。其他人看到他们的成果觉得棒极了，而创造者本人看到的都是自己作品的缺陷。这种视角的差异并非偶然，因为只有对现状不满，才会造就杰出的成果。

如果你能平衡好希望和担忧，它们就会推动项目前进，就像自行车在保持平衡中前进一样。在创新活动的第一阶段，你不知疲倦地猛攻某个难题，自信一定能够解决它。到了第二阶段，你在清晨的寒风中看到自己已经完成的部分，清楚地意识到存在各种各样的缺陷。此时，只要你对自己的怀疑没有超过你对自己的信心，就能够坦然接受这个半成品，心想不管多难我还是可以把剩下的部分做完。

让这两股相反的力量保持平衡是很难的。初出茅庐的年轻黑客都很乐观，自以为做出了伟大的产品，从不反思和改进。上了年纪的黑客又

太不自信，甚至故意回避一些挑战性很强的项目。

任何措施，只要能让“再设计”周而复始地进行下去，就都是可取的。文章可以修改到你满意为止，但是软件的修改通常来说可以无休止地进行下去。文章的读者不可能抱怨修改后新增加的内容让他们前后的思想产生了不协调，但是软件的使用者就会抱怨修改后的版本有不兼容问题。

用户是一把双刃剑。他们推动语言的发展，但也使得你不敢对语言进行大规模改造。所以，一开始的时候要精心选择用户，避免使用者过快增长。发展用户就像一种优化过程，明智的做法就是放慢速度。一般情况下，用户比较少意味着你任何时候都可以加大修改的力度。这时，对语言规格做出改变就像撕绷带，当你感到痛苦的一瞬间，痛苦就已经成为了回忆。如果用户数量庞大，修改语言带来的痛苦就将持续很长时间。

大家都知道，让一个委员会负责设计语言是非常糟糕的主意。委员会只会做出恶劣的设计。但是我觉得，委员会最大的问题在于他们妨碍了“再设计”。在委员会的主持下，修改一种语言是非常麻烦的事，没有人愿意自讨苦吃。而且，即使大多数成员不喜欢某种做法，委员会最后的决定往往还是维持现状。



就算委员会只有两个人，还是会妨碍“再设计”，典型例子就是软件内部的各个接口由不同的人负责。这时除非两个人都同意改变接口，否则接口就无法改变。因此现实中，尽管软件功能越来越强大，内部接口却往往一成不变，成为整个系统中拖后腿的部分。

一种可能的解决方法是，将软件内部的接口设计成垂直接口而不是水平接口。这意味着软件内部的模块是一个个垂直堆积起来的抽象层，层与层之间的接口完全由其中的一层控制。如果较高的一层使用了较低的一层定义的语言，那么接口就由较低的一层控制；如果较低的一层从属于较高的一层，那么接口就由较高的一层控制。

梦寐以求的编程语言

让我们试着描述黑客心目中梦寐以求的语言来为以上内容做个小结。这种语言干净简练，具有最高层次的抽象和互动性，而且很容易装

备，可以只用很少的代码就解决常见的问题。不管是什么程序，你真正要写的代码几乎都与你自己的特定设置有关，其他具有普遍性的问题都有现成的函数库可以调用。

这种语言的句法短到令人生疑。你输入的命令中，没有任何一个字母是多余的，甚至用到 Shift 键的机会也很少。

这种语言的抽象程度很高，使得你可以快速写出一个程序的原型。然后，等到你开始优化的时候，它还提供一个真正出色的性能分析器，告诉你应该重点关注什么地方。你能让多重循环快得难以置信，并且在需要的地方还能直接嵌入字节码。

这种语言有大量优秀的范例可供学习，而且非常符合直觉，你只需花几分钟阅读范例就能领会应该如何使用此种语言。你偶尔才需要查阅操作手册，它本身很薄，里面关于限定条件和例外情况的警告寥寥无几。



这种语言的内核很小，但很强大。各个函数库高度独立，而且和内核一样经过精心设计，它们都能很好地协同工作。语言的每个部分就像精密照相机的各种零件一样完美契合，不需要为了兼容性问题放弃或者保留某些功能。所有函数库的源码都很容易得到。这种语言能够很轻松地与操作系统和用其他语言开发的应用程序对话。

这种语言以层的方式构建。较高的抽象层透明地构建在较低的抽象层之上。如果需要的话，你可以直接使用较低的抽象层。

除了一些绝对必要隐藏的东西，这种语言的所有细节对使用者都是透明的。它提供的抽象能力只是为了方便你的开发，而不是为了强迫你按照它的方式行事。事实上，它鼓励你参与它的设计，给你提供与语言创造者平等的权力。你能够对它的任何部分加以改变，甚至包括它的语法。它尽可能让你自己定义的部分与它本身定义的部分处于同等地位。这种梦幻般的编程语言不仅开放源码，更开放自身的设计。

15

设计与研究

外国游客常常惊讶地发现，美国人交谈的时候，一开始总是问“你干什么工作”。我一直讨厌回答这个问题，因为一句话说不清楚。不过我最终找到了解决方法，现在如果有人问我干什么工作，我会正视对方的双眼说：“我正在设计一种 Lisp 语言的新方言。”如果你也有同样困扰，我推荐你也如此回答。对方就立刻转向其他话题了。

我确实是在“设计”一种编程语言，而且我不认为自己在做“研究”。我所做的工作与其他人设计一幢大楼、一把椅子、一种新字体并没有本质不同。我的目的不是发现一种“新”东西，而是做出一种很“好”的编程语言。

设计与研究的区别看来就在于，前者追求“好”（good），后者追求“新”（new）。优秀的设计不一定很“新”，但必须是“好”的；优秀的研究不一定很“好”，但必须是“新”的。我认为这两条道路最后会发生交叉：只有应用“新”的创意和理论，才会诞生超越前人的最佳设计；只有解决那些值得解决的难题（也就是“好”的难题），才会诞生最佳研究。所以，最终来说，设计和研究都通向同一个地方，只是前进的路线不同罢了。

如果把创造一种编程语言看成是设计问题，而不是科研方向，那么有何不同？

最大的不同在于你会更多地考虑用户。设计的时候，一开始总是问：我为谁设计？他们需要什么？比如，优秀的建筑师不会先设计，然后强迫用户接受，而是先研究最终用户的需求，然后做出用户需要的设计。



注意，我说的是“用户需要的设计”，而不是“用户要求的设计”。我不想让读者产生一种印象，认为设计师就像厨师一样，顾客点什么菜就一模一样做出来。艺术的各个领域有着巨大的差别，但是我觉得任何一个领域的最佳作品都不可能由对用户言听计从的人做出来。

有一句话说“顾客永远是对的”，这是指评价优秀设计的标准是看它能够多大程度上满足用户的需求。如果你的小说没人爱看，或者你做的椅子极不舒服，那么就说明你的作品失败了，被一票否决了。就算你的小说（或者椅子）有着最先进的理论指导也无济于事。

可是，让用户满意并不等于迎合用户的一切要求。用户不了解所有可能的选择，也经常弄错自己真正想要的东西。做一个好的设计师就像做一个好医生一样。你不能头痛医头，脚痛医脚。病人告诉你症状，你必须找出他生病的真正原因，然后针对病因进行治疗。

大多数优秀设计都是这样产生的，它们关注用户，并且以用户为中心。



我说设计必须考虑用户的需求，这里的“用户”并不是指所有普罗大众。事实上，你可以选择任何想要的目标用户。比如，假定你正在设计一种工具，你可以把目标用户定为初学者，也可以定为专家级用户。一种人眼里的优秀设计可能在另一种人眼里却是糟糕无比。这里的重点是你必须选出某些人作为你的目标用户。我觉得，除非设定目标用户，否则一种设计的好坏根本无从谈起。

如果目标用户群体涵盖了设计师本人，那么最有可能诞生优秀设计。如果目标用户与你本人差别很大，你往往会假定目标用户的需求比你本人的需求更简单，而不是更复杂。低估用户（即使出于善意）一般来说总是会让设计师出错。我怀疑那些设计“公共住宅项目[①]”（housing project）的建筑师根本没想过自己住在里面会是什么感觉。编程语言也有这种现象。C、Lisp 和 Smalltalk 都是设计者为了自己使用而设计的，而 Cobol、Ada 和 Java 则是为了给别人使用而设计的。

如果你觉得自己在为傻瓜设计产品，那么很可能不仅无法设计出优

①“公共住宅”指的是由政府出资建造的房产，用来出租给低收入家庭居住，类似于廉租房。——译者注

秀产品，而且就连傻瓜也不喜欢你的设计。

不过，就算你的设计针对的是最高端的用户，你也一样是设计给人类使用。研究就不一样了。做数学研究时，你不会只为了方便读者理解而故意选择一种更麻烦的证明方式，你只会选择最直接、最简洁的证明。我想，一般来说科学研究都是这样。科学观点不需要服从人类工程学（ergonomic）。

到了艺术领域，情况就完全变了。设计必须以人为本。设计椅子的时候，你不能只考虑椅子，还必须考虑人体各种千奇百怪的特点，不可能回避掉这一点。所有的艺术都必须迎合人类的兴趣和极限。举例来说，不考虑其他因素时，肖像画就是比风景画更能引发观众的兴趣。文艺复兴时期的经典绘画作品都是画人的，这并非巧合。如果绘画艺术不能用来表现人类本身，那么绘画也不会成为今天这样受推崇的艺术形式了。



不管你喜不喜欢，编程语言也是以人为本的。我怀疑人类的大脑与躯干一样，都有着许多令人琢磨不透的特点。否则为什么有些事情人类特别擅长，而另一些事情人类干起来特别困难。比如，人类似乎不善于处理精细的工作，所以最好还是交给计算机处理。另一方面，如果人类真的擅长和细节打交道，那么我们应该都用机器语言编程才对。

另外，还要记住一点。怎么理解编程语言？你不要把它看成那些已完成的程序的表达方式，而应该把它理解成促进程序从无到有的一种媒介。这里的意思是说，成品的材料和开发时用的材料其实是不一样的。搞艺术的人都知道，这两个阶段往往需要不同的媒介。比如，大理石是一种非常良好、耐用的材料，很适合用于最后的成品，但是它极其缺乏弹性和灵活性，所以不适合在构思阶段用来做模型。

最后写出来的程序就像已经完成的数学证明一样，是一棵经过精心修剪的树木，上面杂乱滋生的树杈都已经被剪去了。所以，评价一种语言的优劣不能简单地看最后的程序是否表达得很漂亮，而要看程序从无到有的那条完成路径是否很漂亮。某种设计使得最后的程序非常漂亮，但是不一定同时具备漂亮的编程过程。比如，我写过一些宏，它们的作用是自动生成另一些宏，它们看上去非常精美优雅，就像一粒粒精细的

宝石。但是，开发过程非常丑陋，我就是连续好几个小时不停地试错，而且老实说，至今仍然无法完全确定它们是否百分之百正确。

我们常常采用错误的方法评价编程语言，只看一眼最后完成的程序就做出判断。同一个软件有两种不同语言开发的版本，你发现其中一个版本比另一个版本短得多，于是非常自信地认定前者的编程语言比后者的更好。但是，如果你从艺术创作的角度思考这个问题，就不太可能这样评价编程语言。因为你不想最后只剩下一种像大理石那样漂亮、又像大理石那样难用的编程语言。

比如，开发软件的时候，一个“交互式顶层解释器”（interactive toplevel）会带来巨大的优势。在 Lisp 语言中，这种解释器就叫做“读取–求值–打印”循环（read-eval-print loop）。有了这个解释器后，语言的设计就会受到巨大影响。静态类型语言不适合部署这样的解释器，因为静态类型语言要求在使用变量前先声明类型，这对于“交互式顶层解释器”行不通。当你在解释器中输入表达式，然后对变量 x 进行赋值，接着再对 x 做进一步处理时，你只想尽快看到结果，肯定不想很麻烦地先声明 x 的类型。你也许不同意“交互式顶层解释器”为软件开发带来便利的说法，但是如果你接受它，同意易于使用的编程语言必须有一个这样的解释器，那么强制声明变量类型的做法就是与这个解释器不兼容，因此结论就是所有的静态类型语言都不易于编程。



为了做出优秀的设计，你必须贴近用户，始终寸步不离，永远站在用户的角度调整自己的构想。19 世纪英国作家简·奥斯汀的小说为何如此出色？一个原因就是她把自己的作品大声读给家人听，所以她就不会陷入孤芳自赏难以自拔的境地，不会长篇累牍地赞叹自然风光，也不会滔滔不绝地宣扬自己的人生哲学。（事实上，简·奥斯汀还是在小说里宣扬了自己的人生哲学，不过她把它编进故事之中，而不是直接像贴标签那样讲出来。）你可以随便找一本平庸的“文学”读物，想象一下把它当作自己的作品读给朋友们听，这样会让你真切地感受到那些“文学”读物高高在上的视角，读者必须承受所有沉重的负担才能阅读这些作品。

在软件领域，贴近用户的设计思想被归纳为“弱即是强”（Worse is

Better）模式[1]。这个模式实际上包含了好几种不同的思想，所以至今人们还在争论它是否真的成立。但是，其中有一点是正确的，那就是如果你正在设计某种新东西，就应该尽快拿出原型，听取用户的意见。

与之对照，还有另一种软件设计思想，也许可以被称为"万福玛丽亚"模式。它不要求尽快拿出原型，然后再逐步优化，它的观点是你应该等到完整的成品出来以后再一下子隆重地推向市场，就像圣母玛丽亚降临一样，哪怕整个过程漫长得像橄榄球运动员长途奔袭、达阵得分也没有关系。在互联网泡沫时期，无数创业公司因为相信了这种模式而自毁前程。我还没听说过有人采用这种模式而获得成功。

软件领域以外的人可能没听过"弱即是强"，所以意识不到这种模式在艺术领域普遍存在。以绘画为例，文艺复兴时期就有人发现了这一点。如今，几乎所有的美术老师都会告诉你准确画出一个事物的方法，不是沿着轮廓慢慢一个部分、一个部分地把它画出来，因为这样的话各个部分的错误会累积起来，最终导致整幅画失真。你真正应该采用的方法是快速地用几根线画出一个大致准确的轮廓，然后再逐步地加工草稿。

在大多数艺术领域，原型使用的材料与成品的材料一般来说是不一样的。印刷活字先画在纸上，然后才做成铅字。雕塑先用石蜡创作，然后才用青铜浇铸。地毯图案先用墨水画出纸型，然后才织成地毯。建筑物先做出木模型，然后才做成石头建筑。

为什么15世纪油画首次亮相会引起轰动并很快流行起来？原因就是油彩使得画家可以在原型上直接画出最后的样子。你可以按照自己的想法画出初稿，但是它并不对你构成限制。接下来你可以逐步加上细节，甚至对初稿做出重大修改，直到最后完成。

软件开发也可以这样做。原型（prototype）并不只是模型（model），



---

① "弱即是强"指的是一种软件传播的模式，由Common Lisp专家理查德·加布里埃尔（Richard P. Gabriel）于1991年在Lisp: Good News, Bad News, How to Win Big（http://www.dreamsongs.com/WIB.html）一文中首先提出。它的含义非常广泛，涉及软件设计思想的各个方面，其中的一个重要结论就是软件功能的增加并不必然带来质量的提高。有时候，更少的功能（"弱"）反而是更好的选择（"强"），因为这会使得软件的可用性提高。相比那些体积庞大、功能全面、较难上手的软件，一种功能有限但易于使用的软件可能对用户有更大的吸引力。加布里埃尔本人经常举Unix和C语言的例子，Unix和C在设计上考虑了实际环境，放弃了一些功能，但是保证了简单性，这使得它们最终在竞争中胜出，成为主流操作系统和编程语言。——译者注

不等于将来一定要另起炉灶，你完全能够在原型的基础上直接做出最后的成品。我认为，只要有可能，你就应该这样做。这样的方式使得你可以利用在开发过程中一路产生的新想法。不过更重要的是，这样做有助于鼓舞士气。

士气是设计的关键因素。令我吃惊的是，大家很少提到这一点。我的一位美术启蒙老师告诉我：如果你觉得画某样东西很乏味，那么你画出来的东西就会真的很乏味。比如，假设你必须画一幢建筑物，你决定从每一块砖头开始画起。你觉得自己可以坚持下去，但是画到一半的时候突然感到很厌倦，于是你就不再认真观察每块砖头并画出它们各自不同的特点，而是以一种机械重复的方式草草地把砖头画完了事。这样一来，你的作品效果就很差，甚至还不如一开始就不采用写实手法，只是若隐若现地暗示砖头的存在。



先做出原型，再逐步加工做出成品，这种方式有利于鼓舞士气，因为它使得你随时都可以看到工作的成效。开发软件的时候，我有一条规则：任何时候，代码都必须能够运行。如果你正在写的代码一个小时之后就可以看到运行结果，这好比让你看到不远处就是唾手可得的奖励，你因此会受到激励和鼓舞。其他艺术领域也是如此，尤其是油画。大多数画家都是先画一个草图，然后再逐步加工。如果你采用这种方式，那么从理论上说，你每天收工的时候都可以看到整体的效果，不会对最后的成品一点感觉都没有。跟你说实话吧，画家之间甚至流传着一句谚语："画作永远没有完工的一天，你只是不再画下去而已。"这种情况对于第一线的程序员真是再熟悉不过了。

士气也可以解释为什么很难为低端用户设计出优秀产品。因为优秀设计的前提是你自己必须喜欢这种产品，否则你不可能对设计有兴趣，更不要说士气高昂了。为了把产品设计好，你必须对自己说："哇，这个产品太棒了，我一定要设计好！"而不是心想："这种垃圾玩意，只有傻瓜才会喜欢，随便设计一下就行了。"

设计意味着做出符合人类特点和需要的产品。但是，"人类"不仅包括用户，还包括设计师，所以设计工作本身也必须符合设计师的特点和需要。

志　谢

我首先要感谢的人是 Sarah Harlin。每当写完一篇文章，我通常会先给她看。她总是会删去一半，然后告诉我重写另一半。她对文章节奏的把握堪称完美，不放过任何一句废话，就像猎狗不放过麻雀一样。

如果本书写得不算糟，那主要归功于我与她之间的交流以及我与 Robert Morris、Trevor Blackwell、Jackie McDonough 之间的交流。我很幸运，能够认识他们。

其他一些朋友的思想也让我受益匪浅。本书涉及的主要问题都是在过去几年中与他们深入讨论过的。这些朋友是 Ken Anderson、Chip Coldwell、Matthias Felleisen、Dan Friedman、Daniel Giffin、Shiro Kawai、Lisa Randall、Eric Raymond、Olin Shivers、Bob van der Zwaan 和 David Weinberger。我尤其感谢 Eric Raymond，不仅因为他的思想令我受教，还因为他关于黑客的文章[①]为我提供了如何写这个主题的范例。



我还要感谢许多提供帮助和启发的人。他们是 Jülide Aker、Chris Anderson、Jonathan Bachrach、Ingrid Bassett、Jeff Bates、Alan Bawden、Andrew Cohen、Cindy Cohn、Kate Courteau、Maria Daniels、Rich Draves、Jon Erickson、John Foderaro、Bob Frankston、Erann Gat、Phil Greenspun、Ann Gregg、Amy Harmon、Andy Hertzfeld、Jeremy Hylton、Brad Karp、Shriram Krishnamurthi、Fritz Kunze、Joel Lehrer、Henry Leitner、Larry Lessig、Simon London、John McCarthy、Doug McIlroy、Rob Malda、Julie Mallozzi、Matz、Larry Mihalko、Mark Nitzberg、North Shore United、Peter Norvig、Parmets 夫妇、Sesha Pratap、Joel Rainey、Jonathan Rees、Guido van Rossum、Barry Shein、Sloos 夫妇、Mike Smith、 Ryan Stanley、Guy

① 指Eric Raymond的著名文章“如何成为一名黑客”（How To Become A Hacker），网址是 http://www.catb.org/esr/faqs/hacker-howto.html 。——译者注

Steele、Sam Steingold、Anton van Straaten、Greg Sullivan、Brad Templeton、Dave Touretzky、Mike Vanier、Weickers 夫妇、JonL White、Stephen Wolfram 和 Bill Yerazunis。

本书装帧精美，原因是它完全出自排版大师 Gino Lee 之手，我在这方面什么忙都没帮上。我对 Gino Lee 言听计从，从他身上学到了很多关于图书设计的知识。Chip Coldwell 用了几个小时安排好本书的字体，而 Amy Hendrickson 则是用了数天写出了好几个 LaTex 宏程序，使得文章排版后的格式易于阅读。令人惊奇的是，本书的封面某种意义上是 Robert Morris 设计的，他使用图像处理软件 Gimp 对设计稿做了一些大修改。Gilberte Houbart 巧妙而执着地找到了各种需要的图片，并与分布在全世界各处的相关机构联系，获得了使用授权。

O’Reilly 出版公司的工作人员非常杰出。Allen Noren 对于制作好书的真正兴趣足以使得所有人都能对出版业重新燃起希望。Betsy Waliszewski 对于如何让书籍更畅销的远见卓识被我偷偷地全盘继承。Matt Hutchinson、Robert Romano 和 Claire Cloutier 使得整个书籍的制作过程非常顺利。相比大型出版集团，Tim O’Reilly 这样的出版公司会令出版更精彩。



另外，特别感谢 Jessica Livingston。她的意见让本书在各个方面都得到了提高，从封面到正文无不如此。她始终为我加油打气，这也提高了本书的质量，因为她总是对我说许许多多人想读这本书，迫使我只能更努力地写作，让它不辜负那些人的期待。

我从许多人身上学到了黑客技术，但是让我学会画画的主要只是一个人：Idelle Weber。她是一名优秀老师，特别善于用实例教学。我深深地感谢她和她的丈夫 Julian Weber，他们多年来给予了我无微不至的照顾。

最后，我要感谢我的父母。父亲教给我什么是怀疑主义，母亲教给我如何发挥想象力。有了这样的母亲，我眼前的世界就从黑白变成了彩色。

术 语 解 释

抽象（abstract） 隐藏细节。编程语言越抽象，你写出程序所需的运算步骤就越少，每一步的功能就越强。

Ada 一种面向对象编程语言，20 世纪 70 年代末由一个委员会为美国国防部设计。它的下场与你猜想的一样。

人工智能（AI，Artificial Intelligence） 一个概括性术语，用来描述几种尝试让机器学会思考的技术。其中偏重数学的方法已经取得了一些进展（计算机视觉就是一个例子）。

Algol 一种编程语言，最初在 1958 年由一个委员会设计（不利因素），这个委员会的成员非常聪明（有利因素）。它很少用于编程，但是对于后来的编程语言有着巨大影响。

算法（algorithm） 完成任务的方法。菜谱也算是算法。

字母数字式字符（alphanumeric character） 26 个英文字母和 10 个阿拉伯数字的总称。

应用程序编程接口（API，Application Program Interface） 操作系统或函数库提供的一系列命令，供应用程序与其对话。

APL 一种极其简洁的语言，20 世纪 60 年代初由 Ken Iverson 设计，常用于数值应用（numerical application）。它在现代编程语言中的继承者是 J 语言。

应用程序（application） 不属于系统软件的程序，比如文字处理器。操作系统不属于应用程序。这个词的含义不是很精确。

Arc 一种炒作已久却始终没有亮相的 Lisp 方言[①]。

数组（array） 一种存储数据的方式，教材中又称矩阵（matrix），即一

① 作者本人就是Arc语言的设计者。——译者注



个个用数字编号的存储空间，排列成一个 n 维的集合。

应用服务提供商（ASP，Application Service Provider） 这种软件公司允许用户通过网络使用存放在服务器上的软件，而不是采用传统的方式让用户在自己的电脑上安装软件后使用。

汇编语言（assembly language） 机器语言的一种改进版，对程序员更友好一些。它与机器语言使用同样的命令，但是你可以用更简单方便的名称调用这些命令。

B&D 语言（Bondage & Discipline language） 编程规则极其严格的语言，对程序员的纪律性要求很高，好像被枷锁束缚一样。

带宽（bandwidth） 网络连接时传送数据的速度。

贝叶斯定理（Bayesian） 一种统计推断的方法，又称贝叶斯算法。

二进制（binary） 如果这个词前面有冠词（比如 a binary），指的是软件的目标码。如果没有冠词，指的是采用二进制（而不是日常使用的十进制）作为表示数字的方法，即从最右面一位开始，每一位都代表以 2 为底的乘方（而不是以 10 为底的乘方）。所以，二进制的 101 就相当于十进制的 5。大多数计算机内部采用二进制表示数据，因为两种状态（打开和关闭）的电路比十种状态的电路更容易设计。



位操作（bit manipulation） 对某个内存区域的一些简单转换操作，比如在屏幕上移动窗口就可以通过位操作实现。

代码膨胀（bloatcode） 程序过于冗长，大大超过合理的长度。

块结构（block-structured） 某些语言支持的子区域结构。有了这种语法，就可以实现结构化编程，而不是简单地一条条按顺序执行命令。

Blub 困境（Blub Paradox） 程序员的思想往往会受到自己正在使用的语言的束缚，不相信还存在更强大的语言。

自下而上编程法（bottom-up programming） 一种编程的风格，与早期的“自上而下编程法”（top-down）正好相反，“自上而下编程法”要求你把编程任务分解成一个个更小的单元，“自下而上编程法”要求你先开发最底下的层，然后用底层所定义的“语言”开发上一层，这样直到最高层。这两种编程法可以结合使用。

限制（bound） 受到某种资源的约束，比如 I/O 限制、内存限制、CPU 限制等。

跳转（branch） 机器语言的 goto 命令。

布鲁克斯假说（Brooks's Hypothesis） 程序员一天写出的代码行数是一个常量，与他使用什么语言无关。

bug 程序包含的错误。它的历史甚至比计算机还要悠久。20 世纪早期的百老汇舞台剧就经常说“把 bug 处理了”。

缓冲区（buffer） 一个内存区域，用来保存程序需要的输入数据，或者将程序的输出数据累积起来，到一定数量后再输出。

缓冲区溢出攻击（buffer overflow attack） 参见第 10 章的相关注解。

字节码（byte code） 类似于机器语言的计算机语言，但是不局限于特定的计算机。由于字节码与机器语言很类似，所以很容易开发出字节码解释器，让它把字节码转换成相应的机器语言命令。

C 语言 一种简洁优美的计算机语言，由 Dennis Ritchie 在 20 世纪 70 年代初设计。广泛用于操作系统和路由器的编程。



C++ 一种为 C 语言加入面向对象编程能力的语言，由 Bjarne Stroustrup 在 1983 年设计。它流行起来的原因是它的语法与 C 语言很相似，而且用它开发的程序能够与 C 语言程序混合在一起。

CGI 脚本（Common Gateway Interface script，通用网关接口脚本） 当网络服务器需要进行某种运算（比如数据库搜索）而不是直接传输现有文件时所运行的一种程序。CGI 脚本的主要缺点是，每次运行只能生成一个页面，无法像桌面软件那样将数据保存在内存中从而实现与用户的不间断对话。

校验和（checksum） 一种用特征值验证文件的方法。特征值通过对文件的所有信息进行某种计算而产生。比如，文件包含的字符数量就是一种特征值（这种方法的效果并不好）。

循环定义（circular definition） 参见“无限循环”（infinite loop）。

类（class） 面向对象编程语言的一种数据类型。

点击轨迹（click trail） 同一个用户向网络服务器发出的一系列 HTTP 请求，基本等同于他浏览的网页顺序。

客户端（client） 一台向服务器发出请求的电脑或其他设备。

Cobol 一种较原始的编程语言，20 世纪 60 年代初诞生，供商务性应用程序使用。它一度是最流行的语言，直到最近才被 Java 语言取代。

代码（code） 一般情况下指的就是源代码（source code）。

托管（collocated） 通常指放在 ISP 处。

注释（comment） 程序中不被计算机执行的部分，通常是为了向人类读者做说明而插在源码中。

Common Lisp Lisp 语言的一种流行的方言，20 世纪 80 年代由一个委员会设计。

编译器（compiler） 一种程序，将更强大、更流行的语言（高级语言）写的程序翻译成计算机硬件能够理解的命令（机器语言）。参见“解释器”（interpreter）。



复杂度（complexity） 算法的“时间复杂度”（time complexity）指的是，当输入的数据量不断增加时，计算机完成这种算法所消耗的时间。比如，假定你要在一间屋子中寻找某一个人，方法是看每个人的脸，那么找到这个人所需要的时间与屋中的人数成正比。这样一种算法就叫做 O(n)，意为所需的时间与 n 成比例（n 代表数据量）。现在进一步假设你要在屋子中寻找看上去长得很像的两兄弟（或两姐妹），那么你所需要的时间可能与人数的平方成正比，因为你也许不得不每两个人就比较一次，而所有可能的两人组合是人数的平方，算法就是 O(n^2)。

条件结构（conditional） 高级语言的一种表达式（或语句），通过判断条件是否为真而执行不同的代码。比如，判断是否出太阳，如果是的就去散步，否则就待在屋里读书。

基于内容的过滤（content-based filtering） 根据电子邮件的内容而不是它的外部特征（比如发信地址）进行过滤。

CPU（中央处理器，Central Processing Unit） 计算机的一部分，如今通常是一块芯片，负责执行运算。这个概念正在变得模糊，因为处理器现在广泛用于各种各样的设备，比如显卡和硬盘。

崩溃（crash） bug 引起的操作系统或应用程序停止正常工作。用在硬盘上面也指硬件失灵。

冗余（cruft） 多余的不适用的信息（尤其指代码），也可用来指硬件。

周期（cycle） 执行一个机器指令的最少时间。一台内部时钟频率

1 GHz 的计算机可以在 1 秒内完成 10 亿个周期，即每秒执行 10 亿条机器指令。

DARPA（美国国防部高级研究计划局，Defense Advanced Research Projects Agency） 一个赞助美国国内很多计算机研究项目的机构。

数据结构（data structure） 一种由多个部分组成的数据格式。比如，一对数据可以组成一个数据结构，表示图形上的一个点。

数据类型（data type） 编程语言处理的数据种类。典型的数据类型包括整数（比如 1）、浮点数（教材中也称十进制小数，比如 1.234）和字符串（比如 monster）。

动态类型（dynamic typing） 参见“静态类型”（static typing）。

排错（debugging） 发现和纠正程序的错误。

声明（declaration） 程序的要素之一，描述的成分多于命令的成分。最常见的声明是变量类型声明，用于说明一个变量包含哪一类数据。



废弃（deprecated） 原来属于标准的某种做法，现在它的设计者后悔做出了这样的设计。

设计战争（design war） 一种竞争规则，只要是最好的设计就能获胜，而不是其他因素（比如广告宣传、销售渠道的控制）主导竞争。

硬件驱动程序（device driver） 操作系统的一部分，使操作系统可以与硬件设备（比如打印机）对话。

diff 对某件东西的两个版本进行客观的、精细的比较。这个词源自 Unix 操作系统的一个用来比较文件的应用程序。

嵌入式语言（embedded language） 在一种语言内部定义的另一种语言，常用于解决某些特殊的问题。比如，如果你定义了一系列操作图像的命令，你就可以把它们视为一种操作图像的语言，参见“自下而上编程法”（bottom-up programming）。

最终用户（end user） 需求很简单的用户的婉转说法。

编程环境（environment） 帮助编程的软件，比如编辑器和性能分析器。

表达式（expression） 运行后生成一个值的一串代码，比如表达式 2 + 3 将生成 5。

字段（field） 一种数据结构的组成部分。

文件（file） 一串字符或者二进制位，通常存储在硬盘上。

Fortran 一种广泛用于数值应用的编程语言。1956 年，IBM 的一个开发小组设计了这种语言，此后它有了巨大的变化。

FreeBSD 一种 Unix 的开源版本。

自由软件（freeware） 自由传播的软件。

函数（function） 一个子程序，调用后生成一个值返回。在一些语言中函数也是一种数据类型。

垃圾回收机制（garbage collection） 程序自动判断哪些内存不再需要，并予以回收，而不是要求程序员在使用完毕后明确声明（这样的声明经常是错的）。



胶水程序（glue program） 在应用程序之间整理或者转移数据的程序。

goto 将程序的运行顺序从一部分改向另一部分的命令。goto 与子程序调用最大的不同在于它使用后没有办法再回到原处。所以如果用了 goto，程序往往会乱成一团。现在已经很少使用这个命令了。

格林斯潘第十定律（Greenspun's Tenth Rule） “任何 C 或 Fortran 程序复杂到一定程度之后，都会包含一个临时开发的、只有一半功能的、不完全符合规格的、到处都是 bug 的、运行速度很慢的 Common Lisp 实现。”

破解（hack） 一种破坏规则的解决方法，可能有益也可能有害。

黑客（hacker） 解释一，优秀程序员。解释二，侵入他人电脑的人。

散列表（hash table） 一种类似数据库的数据结构，存储在里面的每一段数据都有一个对应的键，使用时只要按照键就可以取出对应的数据。

邮件头（header） 电子邮件最前面的那部分，包含了邮件本身的相关信息。普通用户只会看到发信人（From）、收信人（To）、日期（Date）、主题（Subject）、抄送（Cc）等信息，但是还包含其他信息（比如邮件的传送路径）。

启发（heuristic） 从经验法则中得到灵感。

高级语言（high-level） 比机器语言抽象得多的语言。

HTML（超文本标记语言，HyperText Markup Language） 用来撰写网页的一套书写法。

HTTP（超文本传输协议，HyperText Transfer Protocol） 网络服务器与浏览器之间的通信协议。

缩进（indented） 源代码就像一篇文章的大纲一样，用缩进显示自身的结构。比如，代码的某个部分需要重复 *n* 次，那么这部分代码往往就缩进，表示它们处在一个循环之中。对于大多数语言来说，缩进只是让代码可读性更好。但是，对于另一些语言（比如 Python），缩进有着特别重大的意义，会影响到程序的行为。

无限循环（infinite loop） 参见“循环定义”（circular definition）

解释器（interpreter） 解释器类似于编译器，处理使用高级语言写出的程序。但是，它不是将整个程序转为机器语言再运行，而是每次读入一行代码，然后执行相应的机器语言命令，之后再去读下一行代码。



内循环（inner loop） 一个程序中执行次数最频繁的部分。

记录仪（instrument） 修改程序使得它的每一步结果都得到记录，这样的话，如果程序运行速度缓慢或者占用太多内存，你就能找到原因。

Intel 机（Intel box） 装备 Intel 处理器的计算机。

I/O 输入（Input）和输出（output）。通常是指读取和显示（或打印）字符或二进制数据。

IT（信息技术，Information Technology） 计算机的基础设施以及那些负责维护它们的人。这个词主要是大公司或者非技术行业的公司在用。

Java 由 James Gosling 设计，原意是对 C++进行改进。一开始它的名字叫做 Oak，后来被 Sun 公司改成 Java，他们采用它的目的是，希望在操作系统和应用程序之间加入一个由 Sun 公司控制的层。这个目的没有实现，但是 Java 最终还是流行起来了，一部分原因是 Sun 公司巨大的市场推广投入，另一部分原因是市场确实需要一种比 C++更好的语言。

JavaScript 一种针对浏览器的脚本语言，由 Brendan Eich 设计。它与 Java 语言并没有内在联系，而且本身在很多方面都有设计缺陷。它的名声也不太好，因为许多网站用它干见不得人的勾当。

蹩脚（kludge） 水平很差的破解。（这个词与 stooge“丑角”是押韵的！）

雏形创业公司（larval startup） 创业公司最早期、还未成形的阶段。此时，潜在的创业者还不确定是否应该成立一家公司创业。

历史遗留软件（legacy software） 这些软件虽然还有人使用，但是并不符合使用者的要求。他们继续使用它们只是因为无钱购买新软件，或者不敢改变现状。

闭包（lexical closure） 一个函数，通过它可以引用由包含这个函数的代码所定义的变量。第 13 章附录中的累加器生成就用到了闭包。

LFSP（聪明人的语言，Language For Smart People） 设计目标主要是追求功能强大而不是保证安全性的编程语言。

函数库（library） 已经写好的代码片段，可以用来执行特定任务。

Linux Unix 操作系统的一个开源版本。讲究一点的话，它应该被称为 GNU Linux，因为它的内核部分是由 Linus Torvalds 编写的，但是其他更大量的代码来自 Richard Stallman 的 GNU 项目。



Lisp 一种编程语言的分支，由 John McCarthy 在 20 世纪 50 年代末的研究成果衍生而来。它的两种最著名的方言是 Common Lisp 和 Scheme。如今的开源编程语言越来越多地借鉴 Lisp 的设计。

列表（list） 一连串的数据块，各个数据块的类型通常是不一样的。不同的列表可以像火车车厢一样连接在一起，组成更大的列表。

字面量（literal representation） 一种直接在高级语言中表示数据的方法。大多数语言中，五的字面量是 5。（表达式 2 + 3 也能得到这个值，但是 2 + 3 属于表达式，不是字面量。）

低层次编程语言（low-level） 抽象程度较低的编程语言。它允许使用一些直接控制硬件的简单命令，比如机器语言就是一种低层次编程语言。

机器指令（machine instruction） 机器语言的一条命令。

机器语言（machine language） 机器指令的一个列表，其中的每个指令都能直接被处理器理解。它也可以理解成机器指令的一个执行序列。

宏（macro） 一个能够生成其他程序的程序。要在不同语言中实现这一点，就意味着不同语言的宏差异很大，一种语言的“宏”可能比另

一种语言的“宏”强大得多。

大型机（mainframe） 根据 20 世纪六七十年代的设计而建造的大型计算机。

对数学家的妒忌（math envy） 担心自己不如数学家聪明。这种焦虑的一个重要体现就是让自己的工作成果带有一种完全不必要的数学味。

元循环（metacircular） 当一种语言的解释器用这种语言本身开发时，就会出现这种情况。与其说这是为了做出这种语言的一种实现，还不如说这是描述语言的一种技巧。

方法（method） 面向对象编程中充当某个类的属性的一个子程序。比如，“圆形”类的一个面积方法可能就是计算圆面积的一个子程序。

模块（module） 一组子程序和变量，它们可以被视为是一个整体。通常情况下，模块外部的代码只能访问模块内部一部分专门对外公开的子程序和变量。



摩尔定律（Moore's Law） 摩尔定律的正式版本是指，一块芯片上的晶体管数量每两年就会翻一番。但是，大多数人提到这个术语时，指的却是处理器的运算速度每 18 个月就会翻一番。很多人认为摩尔定律更像是商业计划，而不是产业发展的规律，毕竟它的提出者 Gordon Moore 是英特尔公司的创始人之一。

数字密集运算（number crunching） 对巨量的数值资料进行直接处理。

对象（object） 这个词有很多意思。最常见的涵义是指某种数据类型的一个实例，比如某个特定的字符串，或者某个特定的整数。

目标码（object code） 编译器产生的机器语言。

OO（面向对象，object-oriented） 一种组织程序的方式。假定不同的类代表不同类型的数据，那么针对这些数据执行某种特定任务的代码，可以根据数据的不同被分别写进不同的类，成为这些类的方法。参见第 10 章的介绍。

“奥卡姆剃刀”原则（Occam's Razor） 简单的解释就是较好的解释。

开放源代码（open source） 源代码可以被任何人自由传播和修改的软件，前提条件往往是修改后的软件源代码也必须保持自由状态。Linux 和 FreeBSD 操作系统就是著名的开源软件的例子。

正交的（orthogonal） 彼此独立、能够以多种方式组合在一起的一组东西。经典的乐高积木就比普通的塑料模型玩具更有正交性。

OS（操作系统，Operating System） 控制其他程序运行的程序。Unix、FreeBSD、Linux、Mac OS X 和 Windows 系列都是操作系统。

优化（optimization） 调整程序，使得它的效率更高。

并行计算机（parallel computer） 能够同时执行多个计算任务的计算机。这样的分类并不是很清晰，因为所有的现代处理器为了提高运行速度或多或少都用到了并行处理。

帕金森定律（Parkinson's Law） 完成一项任务所需要的资源会不断扩展，直至把这种资源消耗光为止。

解析器（parser） 读取输入的数据然后生成解析树的程序。

解析树（parser tree） 解析器读取源码后生成的数据结构。它是将源码翻译成机器语言的第一步。



Pascal 从 Algol 语言衍生而来，20 世纪 70 年代初由 Niklaus Wirth 设计。

补丁（patch） 为了修正较早版本的错误而发布的代码。

PDA（个人数字助理，Personal Digital Assistant） 一种可以随身携带的小型计算机。它的操作界面通常比正规计算机更简单，限制也更多。

Perl Larry Wall 开发的一种开源编程语言。最初的设计目的是为了处理字符串，由于这是程序员日常工作的重头戏，所以 Perl 就流行起来了。它的语法复杂（但简炼），它的版本进化快速（但混乱），这两个方面都很出名。

管道（pipe） 将操作系统的各种命令连接起来的一种方式，使得一个命令的输出变成另一个命令的输入。

指针（pointer） 一块数据，它的值是另一块数据的内存地址。

指针运算（pointer arithmetic） 通过对已知地址进行加法运算在内存中找到目标对象。这是低层次语言的一种技巧。

头发高耸的经理（pointy-haired boss） Scott Adams 的连环漫画《呆伯特》中的人物。他象征了那些无能而专横的中层管理人员。

多项式的（polynomial） 用于增长模式时，它表示 y 的增长速度是 x 的乘方，比如 x 的平方或立方。在图形上，乘方越大，y 的增长曲线就越陡峭。

可移植性（portable） （软件）能够被移植到新的硬件上。高级语言程序比机器语言程序更具备可移植性，因为前者的代码（几乎）与硬件无关。

门户（portal） 网站。

过早设计（premature design） 过早决定一个程序的行为。

过早优化（premature optimization） 还没有写完程序，你就开始考虑它的性能问题。这样的软件好比姑娘还没有成年却已经嫁人了。

进程（process） 在同时运行多个程序的操作系统（现代操作系统都具备这种能力）中，同时被运行的程序之一。

编程语言（programming language） （高级语言就是）编译器的输入，然后被转成目标码的东西。（这是开玩笑的说法，详细解释参见第 10 章。）



性能分析器（profiler） 一种观察运行中的目标程序的程序，它会告诉你目标程序的哪一个部分最消耗资源。参见“内循环”。

伪码（pseudocode） 一种不在计算机里而在“纸上”表达算法的语言。有人认为这个概念不过是美化了使用低层次语言时的手工副产品。

Python 一种由 Guido van Rossum 开发的开源编程语言。带有强烈的面向对象风格，被爱好者看作是 Perl 语言的一种较为简洁的替代品。

QA（质量保证，Quality Assurance） 软件行业中负责找出和登记 bug 的人。

递归（recursive） 一种调用自身的算法。警察审讯犯人时就会用到递归。警察先问犯人是否知道案件的情况，或者是否知道谁干的，如果犯人回答知道，那么继续这样问下去。

RAID（冗余独立磁盘阵列，Redundant Array of Independent Disks） 一种硬件，将多个硬盘模拟成单硬盘，（理论上）避免了硬盘崩溃。

“读取–求值–打印”循环（read-eval-print loop） 一种顶层解释器（toplevel）。

正则表达式（regular expression） 一种分解字符串的模式，就像筛子一样从字符串中取出想要的部分。

RISC（精简指令集计算机，Reduced Instruction Set Computer） 这种计算机的机器语言功能有限，但是运行速度较快。它的目标是更好地适应编译器的需要，就像粒度细的胶卷能够拍出更锐利的影像一样。

Ruby 一种较新的开源编程语言，由松本行弘[①]（Yukihiro Matsumoto，又称 matz）开发，是 Perl 和 Python 的竞争者。

扫描（scan） 读取一串字符，将其分解成一个个的语义单位（token）。

Scheme 一种优雅但呆板的 Lisp 方言，由 Guy Steele 和 Gerry Sussman 在 1975 年设计。

脚本语言（scripting language） 一种编程语言，用来对某个程序进行定制。有时，开源编程语言（比如 Perl 和 Python）也被称为脚本语言，但是这种叫法意义不大。



服务器（server） 网络上的一台计算机，用来回应其他计算机的请求。

SETI@home（搜寻地外文明，Search for Extra-Terrestrial Intelligence） 一个科研项目，使用互联网上桌面电脑的空闲计算能力搜索宇宙中其他生命发出的电磁波。

s-表达式（s-expression） 一种语义单位（token），用括号表示，内部可以再包含 0 到多个 s-表达式。

Smalltalk 经典的面向对象的编程语言，由 Alan Kay 在 1972 年设计。

套接字（socket） Unix 操作系统的一种内部渠道，不同计算机的进程通过它可以在网络上交换信息。

软件工程师（software engineer） 程序员的一种正式名称。

面条式代码（spaghetti） 扭曲缠绕在一起的代码，没有人能够读懂，包括作者本人。

spam 无缘无故收到的、数量庞大的垃圾邮件，通常是广告。这个词来自 Monty Python 剧团的喜剧小品，每当餐厅服务员打开一听 Spam 牌

---

① 松本行弘所著的《松本行弘的程序世界》已由人民邮电出版社出版，欢迎购买。新书《代码的未来》也即将出版。——编者注

罐装肉，一群维京海盗打扮的演员就齐声高唱“Spam，Spam，Spam”，将主人公的对话声都淹没了，因此 spam 就有了外界强加的大量打扰的含义。

规格（spec） 软件的规格说明书。对程序功能的一种非正式描述。

SSH（安全 shell，Secure Shell） 可以安全连接远程计算机的一种程序。

SSL（安全套接字层，Secure Sockets Layer） 一种在网络上安全传输数据的协议。

状态机（state machine） 一种理论上的机器，它的所有可能状态是一个集合。当满足某些条件时，状态之间就会发生转换。

语句（statement） 一串不产生值的代码。为了使得自己有用，它必须能够产生一些实际效果，比如显示内容。有人认为这个概念本身就是错的，在一些语言中根本没有语句，只有表达式。



静态类型语言（static typing） 这一类编程语言的所有变量的类型在开始编写程序的时候就必须知道。

字符串（string） 一个字符的序列，常常用 like this 来表示。

子程序（subroutine） 一块独立于其他代码的代码。当程序运行到某个时候，你想使用这块代码，那么直接调用它就可以了。子程序运行结束后，整个程序的控制权会回到调用处。一本食谱中，糖衣的制作方法就可以被看作蛋糕食谱的一个子程序，调用可能是这样表示的：“请按照第 x 页上的方法制作糖衣。”

子集（subset） 某个被其他概念所包含的概念。烘培就是烹饪的一个子集。

西装革履的人士（suits） 非技术人员，尤其指管理层。这个词最早来自他们都穿西装，可是到了 20 世纪 90 年代，他们也开始穿得像黑客一样了。

符号（symbol） 一种数据类型，语义单位（token）就是符号的实例。符号与字符串很类似，除了两点区别：(a) 一个符号就是一个独立的单位，而不仅仅是一串字符；(b) 一个符号通常只有一个对应的名称，而包含同样字符的字符串却可能有好几种。

句法（syntax） 表达程序目的的形式。比如，为 x 赋值 10，不同语言

有不同的写法，`x = 10`，`x <- 10`，或者`(= x 10)`。

系统管理员（system administrator） 安装计算机硬件和软件并保证网络正常运行的人。

系统管理员综合症（system administrator disease） 系统管理员心底里总是认为，他们管理的那些设备只是独立的设备，而不是用户的工具。更概括地说，他们的态度就是将客户视为一种麻烦，而不是自己能够有这个饭碗的原因。这种心理是感受不到竞争压力的工作职位所特有的。

一次性程序（throwaway program） 为了满足暂时性需求而编写的程序。

语义单位（token） 一串同属于一个单位的字符。更通用的叫法是“词”（word）。

顶层解释器（toplevel） 编程语言的一种界面，你可以在其中用这种语言与计算机进行不间断的对话，正如你在Unix的shell界面中所做的那样，而不是简单地编译程序，然后运行，对话就结束了。



树（tree） 一种数据结构，它包含的每一个实例都可以指向两个或更多的实例，比如家谱树。

图灵完备（Turing-complete） 如果一种编程语言写出的所有程序都能被转换成图灵机程序，并且反之也成立，那么这种编程语言就是图灵完备的。所有当代编程语言都是图灵完备的，这意味着（在理论上）它们的功能都是一样强大的。图灵完备又称图灵等价（Turing-equivalent）。

图灵机（Turing machine） 一种完全虚构的计算机，作用是证明计算理论。由于所有计算机的程序都可以被转换成图灵机程序，所以在这个意义上，你不可能做出比图灵机更强大的计算机。但是没有人能保证这一点，因为“计算机”这个词并没有被正式定义过。

类型（type） 即数据类型（data type）。

UDP 一种网络传播信息的协议。

UI 用户界面（User Interface）。

Unix 一种操作系统，当前的大多数操作系统都发源于它。这个词既是一个通用名词，又是一个商标，持有人是一家取得某个早期Unix版本

所有权的公司。20 世纪 70 年代初，这种操作系统诞生于贝尔实验室，开发者是 Ken Thompson 和 Dennis Ritchie。

向量（vector） 一个一维数组，或者一个序列（sequence）。

正常运行时间（uptime） 一台计算机（特别指一台服务器）正常工作的时间百分比。它的另一个意思是指某台计算机从上次宕机到现在的时间长度。

URL（统一资源定位符，Uniform Resource Locator） 网页的地址。更精确地说，是一个指向网络服务器的请求，目标通常是一个网页，但是也可能是要求某种操作（比如网络搜索）。

朦胧件（vaporware） 谈论已久但是迟迟不亮相的软件。

VC（风险投资商，Venture Capitalist） 为他人创业或再融资提供金钱的人，他们要求创业者用股份来交换投资。

1.0 版（version 1.0） 某样东西的第一个版本，暗示不完整、不好用。

VT100 20 世纪 80 年代流行的一种计算机终端。

网络服务器（web server） 一台响应 HTTP 请求的服务器。

楔住（wedged） 表示无回应状态，尤其是指服务器。

所见即所得（wysiwyg） What you see is what you get（你看到的就是你得到的）的缩写（发音为 whizzy wig）。比如文字处理器中，你在屏幕上看到的页面与打印机的输出是一样的。

XML 一种组织数据的格式。



图片授权说明

图2-1，达·芬奇，《女性肖像》，Ailsa Mellon Bruce Fund。© 2004 Board of Trustees，美国国家艺术馆，华盛顿。

图2-2，《蒙特费特罗家族的费德里科》，Copyright Archivo Iconografico S.A./Corbis。

图4-1，摄影 Margret Wozniak。复制得到 Steve Wozniak 许可。



图5-1，版权归 *Popular Electronics* 杂志。复制得到美国计算机历史博物馆许可。

图5-2，复制得到美国阿尔伯克基市警察局许可。

图9-1，摄影 John Colley，复制得到 John Colley 许可。

图9-2，摄影 Alexei Nabarro，图片选自 iStockphoto。

图9-3，英国皇家收藏馆，© 2004伊丽莎白女王二世。

图9-4，复制得到美国宇航局屈莱顿图片收藏室许可。

图9-5，藏于奥地利首都维也纳奥得河畔的维也纳艺术史博物馆。

图13-1，复制得到美国劳伦斯利弗莫尔国家实验室许可。

图13-2，复制得到约翰·麦卡锡许可。

欢迎加入

图灵社区 iTuring.cn

——最前沿的IT类电子书发售平台

电子出版的时代已经来临。在许多出版界同行还在犹豫彷徨的时候，图灵社区已经采取实际行动拥抱这个出版业巨变。作为国内第一家发售电子图书的IT类出版商，图灵社区目前为读者提供两种DRM-free的阅读体验：在线阅读和PDF。

相比纸质书，电子书具有许多明显的优势。它不仅发布快，更新容易，而且尽可能采用了彩色图片（即使有的书纸质版是黑白印刷的）。读者还可以方便地进行搜索、剪贴、复制和打印。

图灵社区进一步把传统出版流程与电子书出版业务紧密结合，目前已实现作译者网上交稿、编辑网上审稿、按章发布的电子出版模式。这种新的出版模式，我们称之为“敏捷出版”，它可以让读者以较快的速度了解到国外最新技术图书的内容，弥补以往翻译版技术书“出版即过时”的缺憾。同时，敏捷出版使得作、译、编、读的交流更为方便，可以提前消灭书稿中的错误，最大程度地保证图书出版的质量。

优惠提示：现在购买电子书，读者将获赠书款20%的社区银子，可用于兑换纸质样书。

——最方便的开放出版平台

图灵社区向读者开放在线写作功能，协助你实现自出版和开源出版的梦想。利用“合集”功能，你就能联合二三好友共同创作一部技术参考书，以免费或收费的形式提供给读者。（收费形式须经过图灵社区立项评审。）这极大地降低了出版的门槛。只要你有写作的意愿，图灵社区就能帮助你实现这个梦想。成熟的书稿，有机会入选出版计划，同时出版纸质书。

图灵社区引进出版的外文图书，都将在立项后马上在社区公布。如果你有意翻译哪本图书，欢迎你来社区申请。只要你通过试译的考验，即可签约成为图灵的译者。当然，要想成功地完成一本书的翻译工作，是需要有坚强的毅力的。

——最直接的读者交流平台

在图灵社区，你可以十分方便地写作文章、提交勘误、发表评论，以各种方式与作译者、编辑人员和其他读者进行交流互动。提交勘误还能够获赠社区银子。

你可以积极参与社区经常开展的访谈、乐译、评选等多种活动，赢取积分和银子，积累个人声望。

TURING 图灵教育

图灵最新重点图书



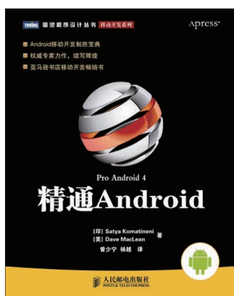
- **计算机体系结构的“圣经”**
- **与时俱进的教科书**
- **全新的第 6 章，讨论仓库级计算机**

本书堪称计算机体系结构的“圣经”，是计算机体系结构方向的学生的必读教材。全书系统地介绍了计算机系统的设计基础、指令集系统结构、流水线与指令级并行技术、层次化存储系统与存储设备、互连网络以及多处理器系统等重要内容，讲述了使用“量化研究方法”进行计算设计，以及多种可以实现并行的技术。在新版本中，增加了涉及移动中心和数据中心的示例，反映了在移动设备和云计算方面的变化；扩展和改进了多核和 CPU 体系结构方面的内容；增加了关于 warehouse-scale 计算机以及向量处理器和 CPU 的内容等等。

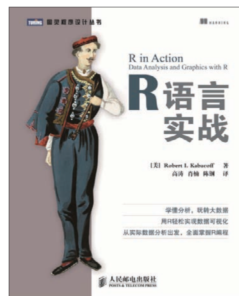
计算机体系结构：量化研究方法（第 5 版）
书号：978-7-115-29765-5
作者：John L. Hennessy David A. Patterson
定价：109.00 元



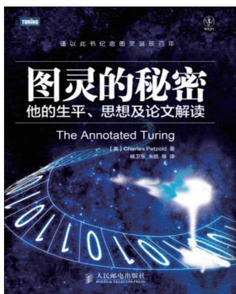
编译器设计（第 2 版）
书号：978-7-115-30194-9
作者：Keith D.Cooper
Linda Torczon
定价：99.00 元



精通 Android
书号：978-7-115-29715-0
作者：Dave MacLean
Satya Komatineni
定价：119.00 元



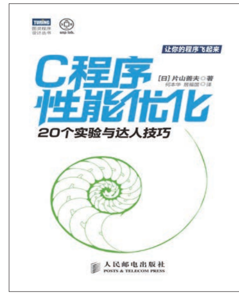
R 语言实战
书号：978-7-115-29990-1
作者：Robert I. Kabacoff
定价：79.00 元



图灵的秘密：他的生平、思想及论文解读
书号：978-7-115-28214-9
作者：Charles Petzold
定价：69.00 元



响应式 Web 设计：HTML5 和 CSS3 实战
书号：978-7-115-29922-2
作者：Ben Frain
定价：49.00 元



C 程序性能优化：20 个实验与达人技巧
书号：978-7-115-30000-3
作者：片山善夫
定价：29.00 元



Paul Graham

Lisp专家，世界上首个互联网应用程序Viaweb开发人之一。创建的Viaweb公司后被雅虎收购，改名为Yahoo!Store。2005年创办Y Combinator，开创了天使投资新模式，被《福布斯》杂志喻为“撼动硅谷的人”。目前为止其公司扶持的创业公司已有250余家，成功的超过80%。Graham是当之无愧的“硅谷创业之父”。其个人网站是：http://www.paulgraham.com。

“此书将迫使你重新思考
计算机编程的本质。”

—— **Robert Morris**，麻省理工学院副教授，
世界上首个互联网应用程序**Viaweb**开发人之一

“人类社会还没有充分理解程序员带来的美和智慧，Graham的这本书却做到了这一点，描述得清晰又动人，任何愿意倾听的人都会大有收获。如果我们不愿意马上就失去一些重要的东西，那么我们这个社会就应该学会倾听。”

—— Lawrence Lessig，斯坦福大学法学院教授

“这是真正睿智之士的思维激荡，读来令人着迷。单单是‘为什么书呆子不受欢迎’一文就值得你买这本书了，它回答了我们这个时代的一个关键问题。”

—— Chris Anderson，《连线》杂志主编

“Paul Graham高瞻远瞩，文笔优雅清晰，且不乏幽默，这在黑客群体中实属罕见，甚至足以跻身优秀作家的行列。”

—— David Weinberger，*Cluetrain Manifesto*作者

“Paul Graham的《黑客与画家》是一本内容广泛的书，但是重要的不是你能从中知道为什么书呆子在高中时备受挫折，或者计算机语言设计和实现有什么奥妙，而是他在论述每一个题目时采取的那种方法，那样生动有趣，富有启迪性，让你莞尔一笑，然后陷入思考。强烈推荐此书给所有读者。”

—— Rob “CmdrTaco” Malda，Slashdot.org创始人、负责人

“Paul Graham是一名黑客，一位画家，还是个出色的作家。他的文章清晰易懂、幽默生动，从艺术、科学、商业互相交织的角度谈论如何写出优秀的代码，充满了与众不同的看法和切实可行的高见。你甚至可能因为看了他的文章而改用Lisp编程哦！”

—— Andy Hertzfeld，苹果机发明人之一

“这是我最近读到的最发人深思的一本书，行文明白流畅，主题多样，而且风趣幽默。”

—— Jeff “hemos” Bates，OSDN负责人、Slashdot.org维护者

图灵网站：www.turingbook.com **热线**：(010)51095186
反馈/投稿/推荐信箱：contact@turingbook.com
有奖勘误：debug@turingbook.com

分类建议 计算机/IT人文

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

O'REILLY®
oreilly.com



ISBN 978-7-115-24949-4
定价：49.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这可以找到我们：

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks